TRSDOS & DISK BASIC Reference Manual



Contents

1. General Information
2. Mini Disk Operation
3. TRSDOS Overview
4. TRSDOS Commands
5. Extended Utilities
6. TRSDOS Technical Information
7. DISK BASIC
8. Appendices

Index

CUSTOM MANUFACTURED IN THE USA FOR RADIO SHACK TC A DIVISION OF TANDY CORPORATION

TRSDOS & DISK BASIC Reference Manual

SCRYPTION
museum voor schriftelijke communicatie

For the Radio Shack TRS-80
Disk Operating System
TRSDOS Version 2.1
DISK BASIC Version 1.1

Radio Shack®
A DIVISION OF TANDY CORPORATION
One Tandy Center
Fort Worth, Texas 76102

First Edition – 1979

All rights reserved. Reproduction or use, without express permission, of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

© Copyright 1979, Radio Shack
A Division of Tandy Corporation,
Fort Worth, Texas 76102, U.S.A.

Software Copyright Notice

All TRSDOS and DISK BASIC software is copyrighted by Radio Shack. Radio Shack grants each TRSDOS user the privilege of making BACKUP diskettes of TRSDOS and DISK BASIC, **provided such diskettes are solely for personal use.**

Any other duplication of TRSDOS or DISK BASIC software, in whole or in part, in print or in any other storage-and-retrieval system, is forbidden.

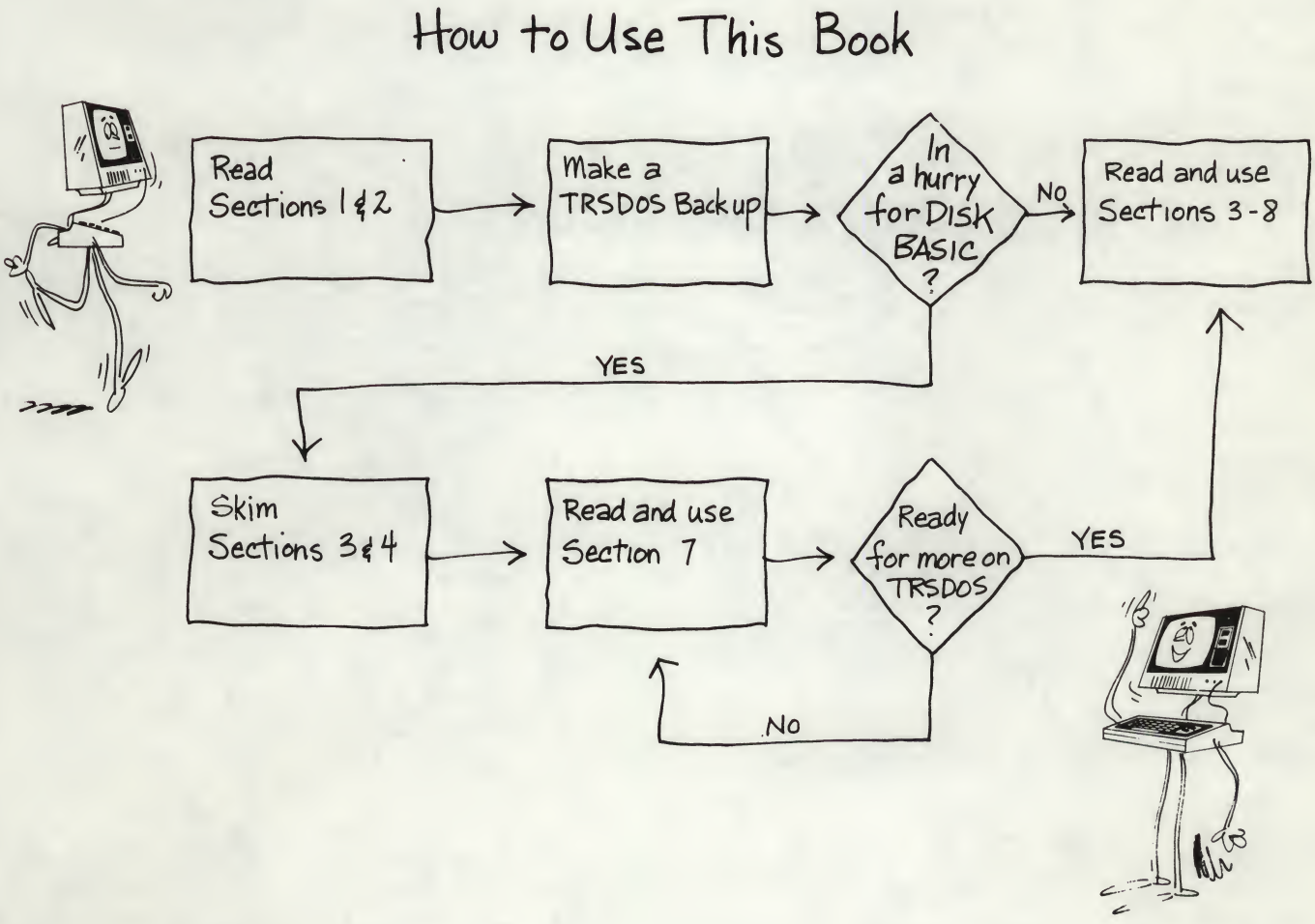
Printed in the United States of America

To Our Customers

This is a reference manual, and its organization reflects the relationship between TRSDOS and DISK BASIC. TRSDOS is the fundamental software, so it's described first. DISK BASIC is a language supported by TRSDOS, so it's described after TRSDOS. (If other languages are supported later, they'll plug right in to this manual along with DISK BASIC.)

But don't think you have to read the manual in strict sequence. If you're an old hand at LEVEL II BASIC and you want to start out with DISK BASIC, go ahead and skip to Section 7. You can refer back to the TRSDOS sections later on when you're ready or when you need them.

We hope you enjoy exploring this powerful new computer system!



General Information



Contents of This Section

Introduction 2
Notation Conventions 3
Versions and Releases 6

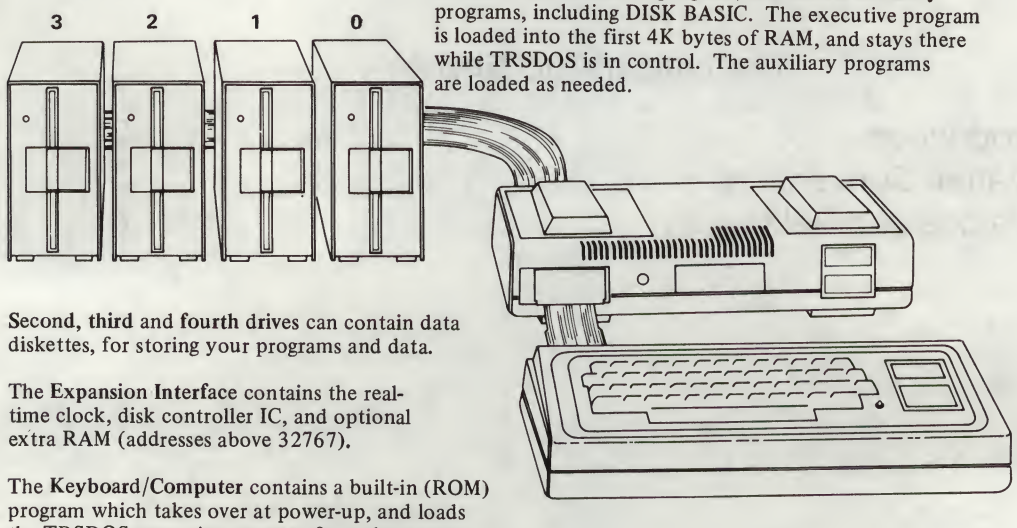
Introduction

This book is a combined operation and reference manual for the TRS-80 Disk Operating System. It will tell you how to operate the hardware and how to use the software.

For many of you, there will be more than enough information. ("All I want to do is use the Computer, not understand it!") Don't worry, this book is designed so that you can start programming in DISK BASIC (if that's what you want to do) right away. All you have to do is read the chapter on **Mini Disk Operation** . . . skim through **TRSDOS Overview** and **TRSDOS Commands** . . . and on to DISK BASIC.

But DISK BASIC is just one aspect of TRSDOS. It's not a part of TRSDOS, but a *program* that TRSDOS executes. Using DISK BASIC without any awareness of the capabilities of TRSDOS is rather like riding in a Pullman car without any knowledge of the engine, freight cars, diner and other parts of a train. It's true that TRSDOS will do all that's necessary to let you ride comfortably along in BASIC; but eventually you're going to want to have a say in where the train goes, what its schedule is, and what goes in all those freight cars. That's when you need to understand TRSDOS.

The illustration below shows the relationship between the Computer, Expansion Interface and Mini Disk Drives.



The **first drive**(Drive 0) always contains the TRSDOS diskette, which is pre-recorded with the Operating System software: an executive program, and several auxiliary programs, including DISK BASIC. The executive program is loaded into the first 4K bytes of RAM, and stays there while TRSDOS is in control. The auxiliary programs are loaded as needed.

Second, third and **fourth drives** can contain data diskettes, for storing your programs and data.

The **Expansion Interface** contains the real-time clock, disk controller IC, and optional extra RAM (addresses above 32767).

The **Keyboard/Computer** contains a built-in (ROM) program which takes over at power-up, and loads the TRSDOS executive program from the system diskette (in Drive 0). If the Mini Disk equipment is not connected, this ROM program can transfer control to LEVEL II BASIC.

One section of this book you should definitely become familiar with is the **Glossary**. We've tried to give definitions for all the "computer words" and everyday words with special meanings in this book. Even if you've heard all the terms, you'll gain some useful information from the **Glossary**, because it's customized for the TRS-80.

First you make a BACKUP . . .

You received one TRSDOS diskette with your Mini Disk drive 26-1160. This diskette contains the operating system software. Without this disk, you haven't got a disk operating system.

So, your first disk operation . . . before you remove the write protect tape from the TRSDOS diskette . . . should be to duplicate TRSDOS onto a blank diskette. You'll find abbreviated instructions for making a duplicate (BACKUP) of your TRSDOS diskette at the end of the **Mini Disk Operation** chapter.

Notation Conventions

In descriptions of syntax for commands, statements and dialog with the Computer, we'll use the following conventions for clarity and brevity.

♭	This special symbol represents a mandatory blank space. Unless it is specified, any blanks that appear in the syntax are optional. Example: DIR♭:1 The blank space is required after the R.
ENTER	"Press the **ENTER** key."
< SPACE >	"Press the space-bar."
CAPITALS and punctuation	Indicate material which must be entered exactly as it appears. The only punctuation symbols not entered are the special cases (brackets and triple-period . . .) explained below. Example: LOAD"*filespec*" Only the command LOAD and the quote marks are entered verbatim; you supply the *filespec.*

Notation, continued

SCREENED CAPITALS — Represent input you supply, upon prompting from the Computer. This convention will only be used where necessary to distinguish between Computer prompting and user input.
Example:

HOW MANY FILES? 5 ENTER

The Computer asks the question, and you answer it.

lowercase italics — Represent words, letters or values you supply from a set of acceptable values for that situation.
Example:
var = exp
A variable name goes on the left, and an expression goes on the right.

[ ] — Brackets enclose optional material.
Example:
CLOSE[*filenum*]
filenum (the file number) is optional after CLOSE. The brackets are not actually typed in.

. . . — The triple-period symbol inside brackets indicates that preceding items in the brackets may be repeated.
Example:
INPUT[*"prompting message"*;]*var*[,*var* . . .]
The INPUT variable-list may include more than one variable. The periods are not actually typed in.

var([,. . .]) — Signifies an array. If no commas are placed inside the parentheses, a one-dimensional array is intended; 1 comma indicates a two-dimensional string array; etc.
Examples:
A$(,) indicates a two-dimensional string array.
B1( ) indicates a single-dimensioned array.

Notation, continued

exp	String or numerical expression
var	String or numerical variable name
nmexp	Numerical expression, including constants, variables, functions
nmvar	Numerical variable name
exp$	String expression, including constants in quotes, variables, functions and operators
var$	String variable name
con	Constant, either string or numerical
nmcon	Numerical constant
con$	String constant
numerical suffixes	Attached to distinguish between different arguments and parameters of the same type. Example: COPY␣*filespec1*␣TO␣*filespec2*

Versions and Releases

Some of you may be a little confused about the terminology, "Version X.Y". The "X" and "Y" will change as TRSDOS is updated, so here's an explanation.

A new version represents a substantial expansion of the previous version. For example, new utilities, high-level languages, etc., might be included in a new version. Such versions are numbered by the integers 1, 2, 3, ...

A new release, on the other hand, is simply an update of the previous release of a given version. This later release generally includes wider implementations and enhancements of commands and fixes for any problems in the earlier release. The releases are numbered by decimal fractions, .1, .2, .3, ...

Therefore, when we refer to Version 2.1, that's short for the first Release of Version 2.

Note: In its original printing, this Manual describes TRSDOS Version 2.1, and DISK BASIC Version 1.1. The Manual will be updated as required by later versions and releases.

Mini Disk Operation



HARDWARE

Contents of This Section

Introduction 2
Connection 3
Operation 5
Care of Diskettes 8
Specifications 10
Schematics 11
Making a TRSDOS BACKUP 16

Note: Abbreviated instructions for making a BACKUP (duplicate) of your TRSDOS software diskette are included at the end of this section. Your very first disk operation should be to make such a "safe copy", following the abbreviated instructions.

Introduction

The TRS-80 Mini Disk drive is a mass storage device custom manufactured for use with the TRS-80 Microcomputer. It combines the compactness of a cassette recorder with the high-speed, reliable data access of the larger disk drive units. Information is magnetically recorded on and read from flexible ("floppy") diskettes.

In simplified terms, the Mini Disk consists of a magnetic read/write head, similar to that on a tape recorder; a stepper motor to move the head across the diskette surface; a drive motor and hub assembly to rotate the diskette; and the necessary logic circuitry to control the read/write process and the motor speed. See Figures 1 and 2.

There are two types of drives, distinguished by their Radio Shack Catalog Numbers, 26-1160 and 26-1161. Your disk system must include one (and only one) 26-1160 and may include up to three 26-1161 drives.

Included with 26-1160

Drive unit: Incorporates special terminating resistors not present in the 26-1161 units.

Interconnect cable: For connection of 26-1160 and up to three optional 26-1161 drives to the Expansion Interface.

1 TRSDOS diskette: Contains the operating system software, utilities, DISK BASIC, etc.

Included with each 26-1161

Drive unit: Does not incorporate terminating resistors.

Blank diskette: Can be formatted or backed up for use with TRSDOS.



Figure 1. Mini Disk Drive.

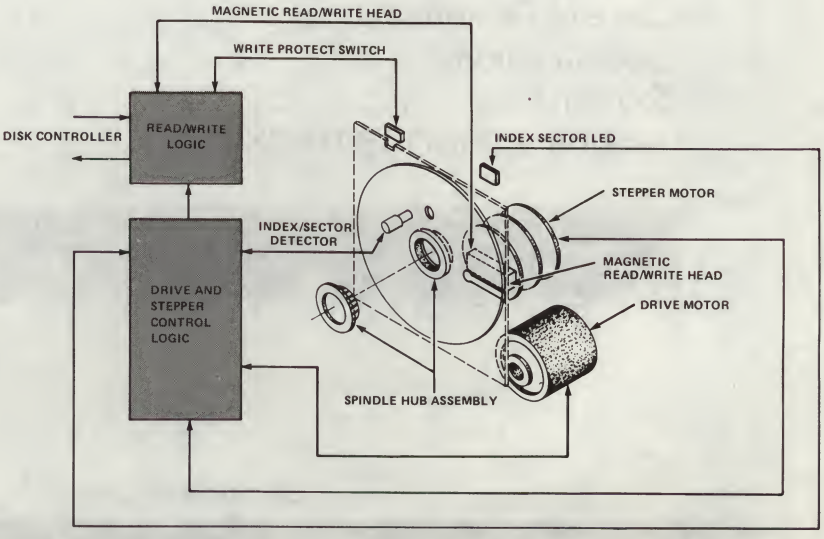


Figure 2. Functional components in a Mini Disk drive.

Connections

The power to all components in the TRS-80 system should be "off" while you make connections.

Look at the ribbon-type connector cable included with your 26-1160 Mini Disk drive. Notice that the cable has four edge card connectors through its length, and a single connector at the other end. Connect the single plug to the edge-card jack on the left rear of the Expansion Interface, as shown in Figure 3. Be sure the plug is oriented so the cable exits from the bottom.

Before connecting the Drive(s) to the cable, note the following rules:

1) 26-1160 must always be the "terminal" or final drive on the cable; that is, of all your drives, it must always be the farthest away from the Expansion Interface. This is because it includes the terminating resistors mentioned above.

2) The connector closest to the Expansion Interface must always be plugged in to a drive. The other connectors can be "empty".

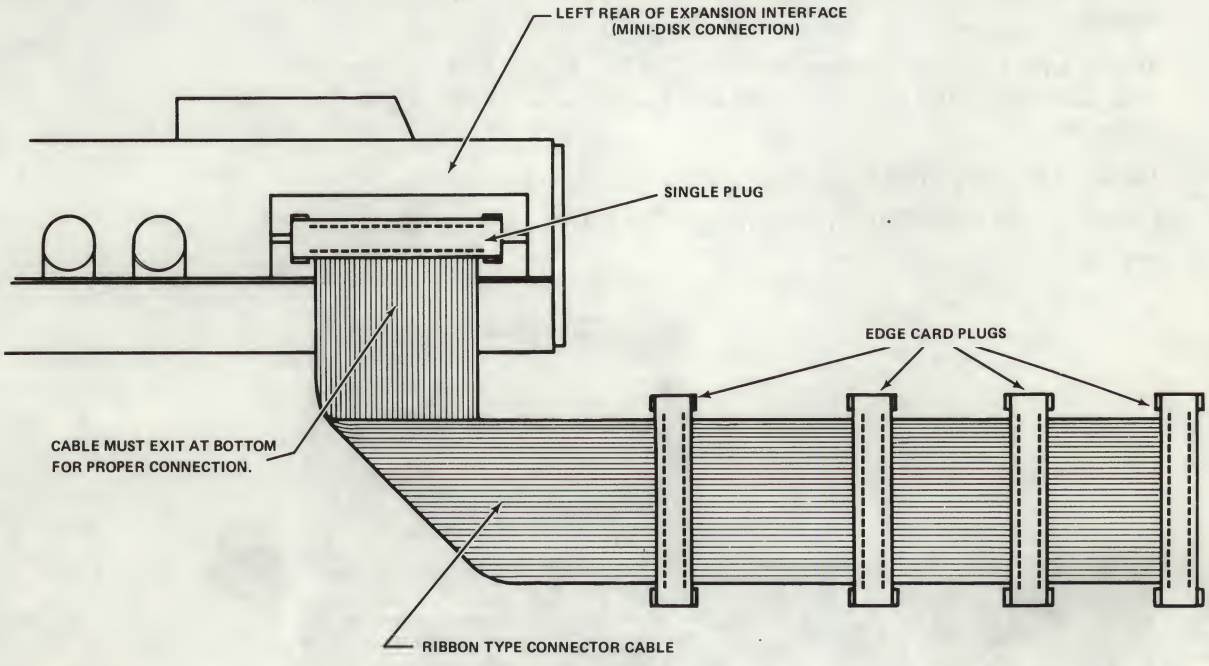


Figure 3. Connecting the ribbon cable to the Expansion Interface.

Connect each Mini Disk unit to the cable, taking care to orient the plug properly as shown in Figure 4. Inside each plug is a small plastic connector. If the plug doesn't mate properly, check to see that the plug is oriented so the pin lines up with the slot.

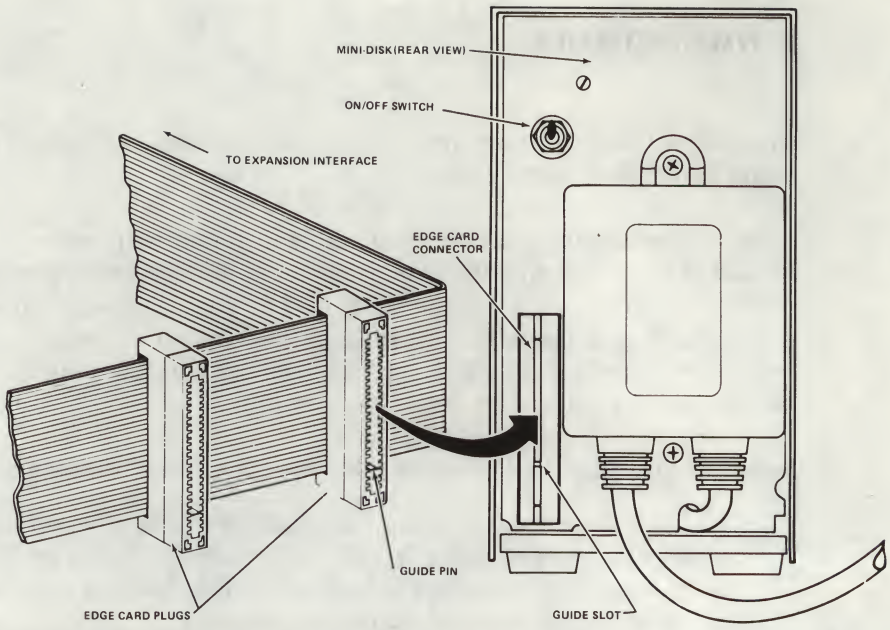


Figure 4. Connecting the cable to the Mini Disk.

Examples:

If you have just one drive (must be 26-1160), then connect it to the first connector plug, so as not to leave any empty connectors between the Drive and the Expansion Interface. Leave the last three connectors empty.

If you have two drives, then connect 26-1161 to the first connector and 26-1160 to the second connector. Leave the last two connectors empty.

Figure 5 shows a Mini Disk system with four drives connected.

Connect each Mini Disk to a source of 120 VAC, using the power cord provided.



Figure 5. A complete four-drive Mini Disk System.

Drive Numbering

TRSDOS requires at least one Mini Disk drive, and can handle up to four. Under TRSDOS, these drives are referred to as drives 0,1,2 and 3 (where drive 0 is closest to the Expansion Interface, and drive 3 is farthest away). See Figure 5. These designations cannot be changed – they are built into the ribbon cable connector.

When the Computer attempts a bootstrap operation (power-on or reset), it will automatically attempt to load TRSDOS from drive 0. Therefore a TRSDOS diskette must be in drive 0 when you power on or reset the Computer. In fact, the TRSDOS diskette should always remain in drive 0 while TRSDOS is in use, except in special cases.

Operation

Before powering on the disk system, you need to understand a few things about how the drives work.

The disk drive does not rotate continuously while it is "on". It only rotates when a Motor-On signal is sent from the Computer. If more than one Mini Disks are connected, the Motor-On signal will turn them all on and off simultaneously, even if only one of them is to be accessed by the Computer. This signal is sent about a second before the Computer accesses the disk, to allow the drives to reach operating speed.

While the Computer is accessing one of the Mini Disks, the red light (LED) on the front of that Mini Disk will remain lit.

Caution: Do not open a drive latch to insert or remove a diskette while the drive motors are running (i.e., while one of the LEDS is lit).

How a Diskette Works

A diskette is simply a circular plastic sheet, one side of which is coated with a highly polished layer of ferromagnetic material. Similar to a 45 RPM record, the diskette has a large spindle hole to accommodate the drive hub, and a small hole which indexes the diskette as it rotates.

A blank diskette (either brand-new or magnetically erased) contains no information. TRSDOS has a special utility program (called FORMAT) which takes a blank diskette and organizes it into concentric "tracks" and subtracks called "sectors". See Figure 6. These divisions are like the numbered pages in a book. (FORMAT also places a small amount of system and bookkeeping information onto each diskette. For more information, see **Extended Utilities,** FORMAT.)

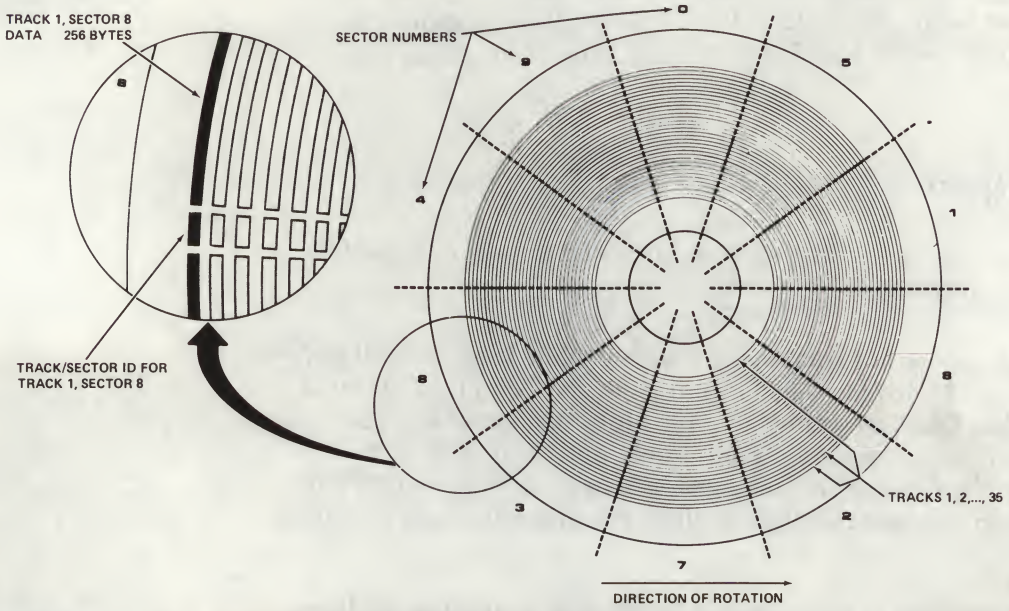


Figure 6. Track/sector organization on a formatted diskette.

Each diskette is permanently sealed inside its jacket to prevent bending, creasing, scratching or contamination of the diskette surface. When the diskette is loaded into the drive, a hub assembly grips the diskette; when the drive motor is on, the diskette rotates inside its jacket. The specially treated jacket lining cleans the diskette as it rotates.

Notice that the TRSDOS diskette comes with a piece of tape across the top (above the label). This tape covers the diskette's write protect notch. With the notch covered, the diskette is physically protected from being written to. (A "write operation" is any alteration of the data stored on the diskette. In contrast, a "read" does not alter the information – merely accesses it.)

Remove the tape from the diskette if you intend to write to it; and place a tape over the notch on any diskette you don't want to accidentally write to.

See Figure 7.

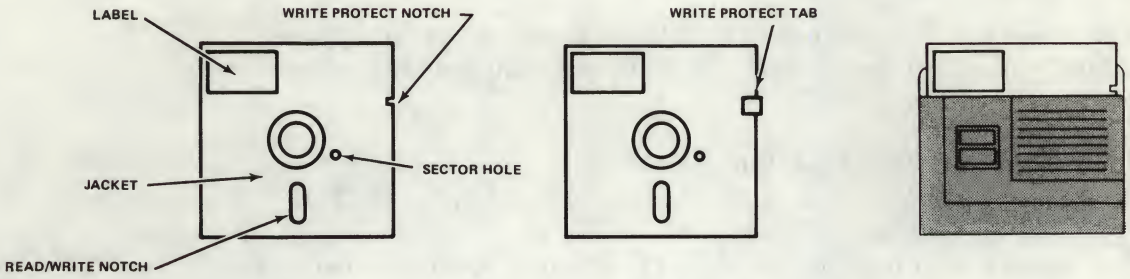


Figure 7. A diskette; a write-protected diskette; a diskette in protective storage envelope.

Inserting a Diskette

1. Be sure the Mini Disk drive is stopped when you insert or remove a diskette.
2. Open the front of the Mini Disk drive. Gently insert the diskette into the vertical slot, with the write protect notch up and the diskette label to the right (Figure 8). Be sure not to close the latch until the diskette is inserted all the way and seated properly, or you may damage it.
3. Close the Mini Disk latch. This causes the spindle-hub assembly to grip the diskette. If the door doesn't close easily, don't force it. Re-insert the diskette and try again.



Figure 8. Inserting a diskette.

Power-Up Sequence

You should always power up the peripherals (disk drives, printer, Expansion Interface, etc.) first, and the TRS-80 CPU/keyboard last. Also note that turning the peripherals on and off while the Computer is on may confuse the system and cause abnormal operation. Work done on a currently open file may be lost.

The power switch for each Mini Disk is on the rear of the unit. Power is "on" when the toggle switch is in the up position, and "off" when the switch is down.

1. Turn on the Expansion Interface.
2. Turn on the Mini Disk drives: first the terminal drive, 26-1160, then the other drives, if any.
3. When you turn on the TRS-80 CPU/keyboard, the Computer will instantly attempt to load TRSDOS from Drive 0. So before turning on the CPU, carefully insert the TRSDOS diskette into drive 0 as explained above under "Inserting a Diskette". You may also want to insert formatted diskettes into the other drives now; however, these may be inserted any time the drives are stopped.

Another approach would be to plug all devices into an adequate power strip and turn them all on with a single switch.

Care of Diskettes

Diskettes are precision recording media. Handle them very carefully to get maximum life from each diskette. In general, follow the special handling precautions used with both tape cassettes and high fidelity records.

1. Keep the diskette in its storage envelope whenever it is not in one of the drives. Don't leave the diskettes in the drives needlessly, for example, when the system is turned off.
2. Keep diskettes away from magnetic fields (transformers, AC motors, magnets, etc.). Strong magnetic fields will destroy information on the diskettes.
3. Handle the diskette by the jacket only – don't touch any of the exposed surfaces. Don't try to wipe or clean the diskette surface; you might scratch it and destroy data.
4. Keep the diskette away from heat and direct sunlight. See the "Specifications" section below for storage temperature range.

5. Avoid contamination of the diskette with cigarette ashes, dust or other particles.
6. Do not write directly on the diskette jacket with a hard-point device such as a ball point pen or lead pencil, as this could damage the recording surface. Use a felt tip pen only.
7. Before inserting a diskette into the Mini Disk drive, be sure the motor is off (no LEDs lit and no motor sound).
8. Store diskettes in a vertical file folder or on a shelf where they are protected from pressure to their sides (just as phono records are stored).

If you have problems . . .

Frequent occurrences of disk I/O errors during disk accesses may indicate a worn diskette or some problem with the Mini Disk drive or other hardware. Try to isolate the problem by swapping drives and diskettes as available.

If you have a repeated problem with a particular diskette, try copying the accessible files onto another diskette. Then erase the faulty diskette with a bulk eraser (Radio Shack Catalog Number 44-210) and attempt to format it (see **Extended Utilities,** FORMAT).

During the format process, the diskette will be checked for flaws, and any defective tracks will be locked out, leaving you with an otherwise usable diskette.

If the Mini Disk drive seems to be at fault (errors during access to several diskettes), bring it in to your local Radio Shack store for servicing.

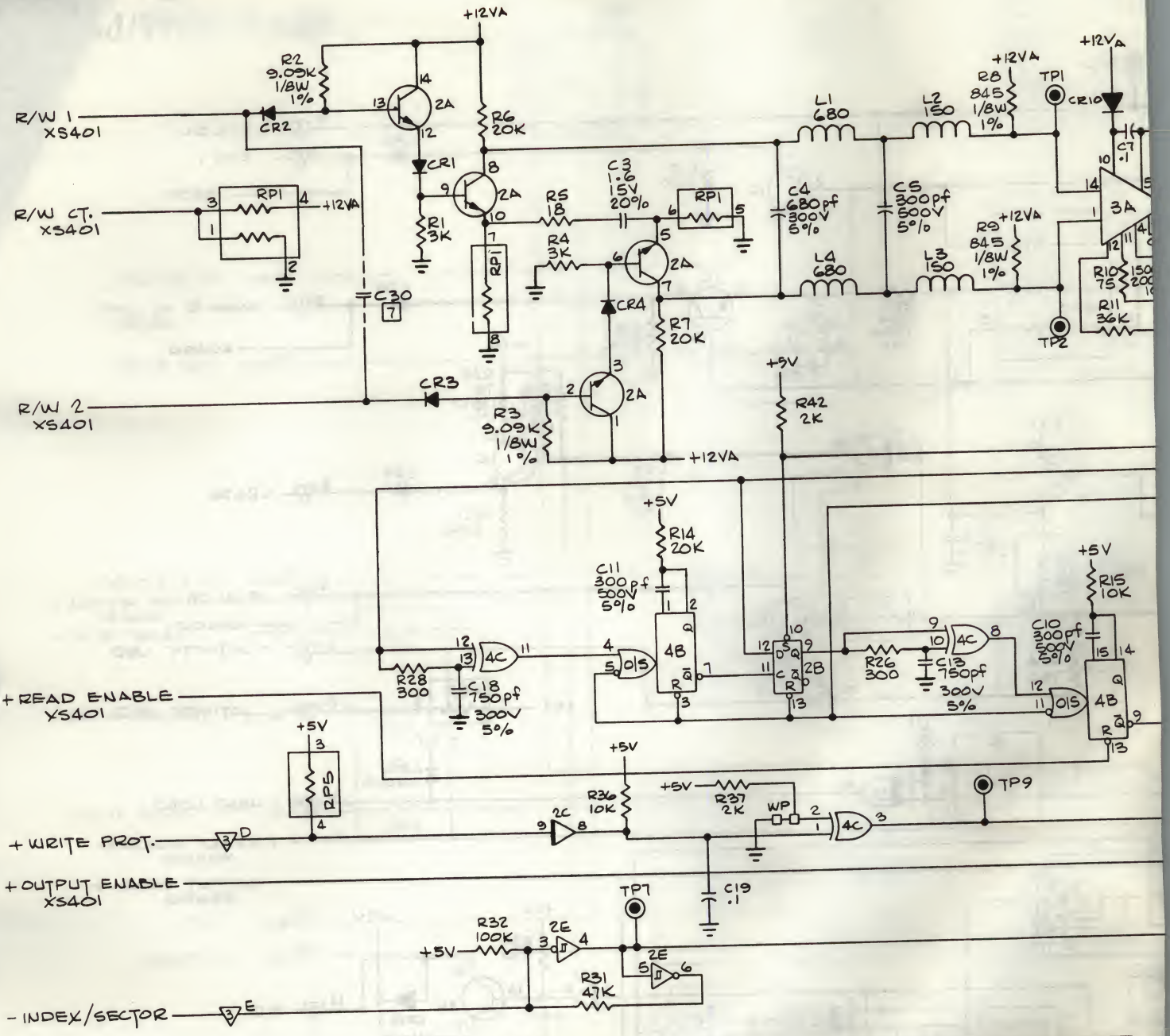
Specifications – Drives and Diskettes

Storage capacity (bytes available to user)	
Formatted diskette	83,060
TRSDOS diskette	58,880
Diskette Organization	
Tracks per diskette	35
Bytes per track	2560
Sectors per track	10
Bytes per sector	256
Data transfer rate	12.5K bytes/second
Average access time	750 mS
Drive motor start time	1 second
Required media	Radio Shack Flexible Diskettes, Catalog Number 26-305, or 26-0405 (pkg of 3)
Diskette life*	2.5×10^6 passes/track (110 hrs) 5 years estimated actual use
Data storage life on diskettes	20 years
Diskette storage temperature	50-125 deg.F (12-52 deg.C)
Size	
Drive unit	6-3/8 x 3-1/2 x 13-1/4" (16.2 x 8.4 x 33.7 cm) HWD
Diskettes (jacket size)	5-1/4 x 5-1/4 x 1/32" (13.3 x 13.3 x 0.08 cm) HWD
Power requirements	120 VAC, 60 Hz, 35 Watts (28 VA)

* Typically, diskette life will be limited by improper handling. Follow handling recommendations listed above for maximum diskette life.

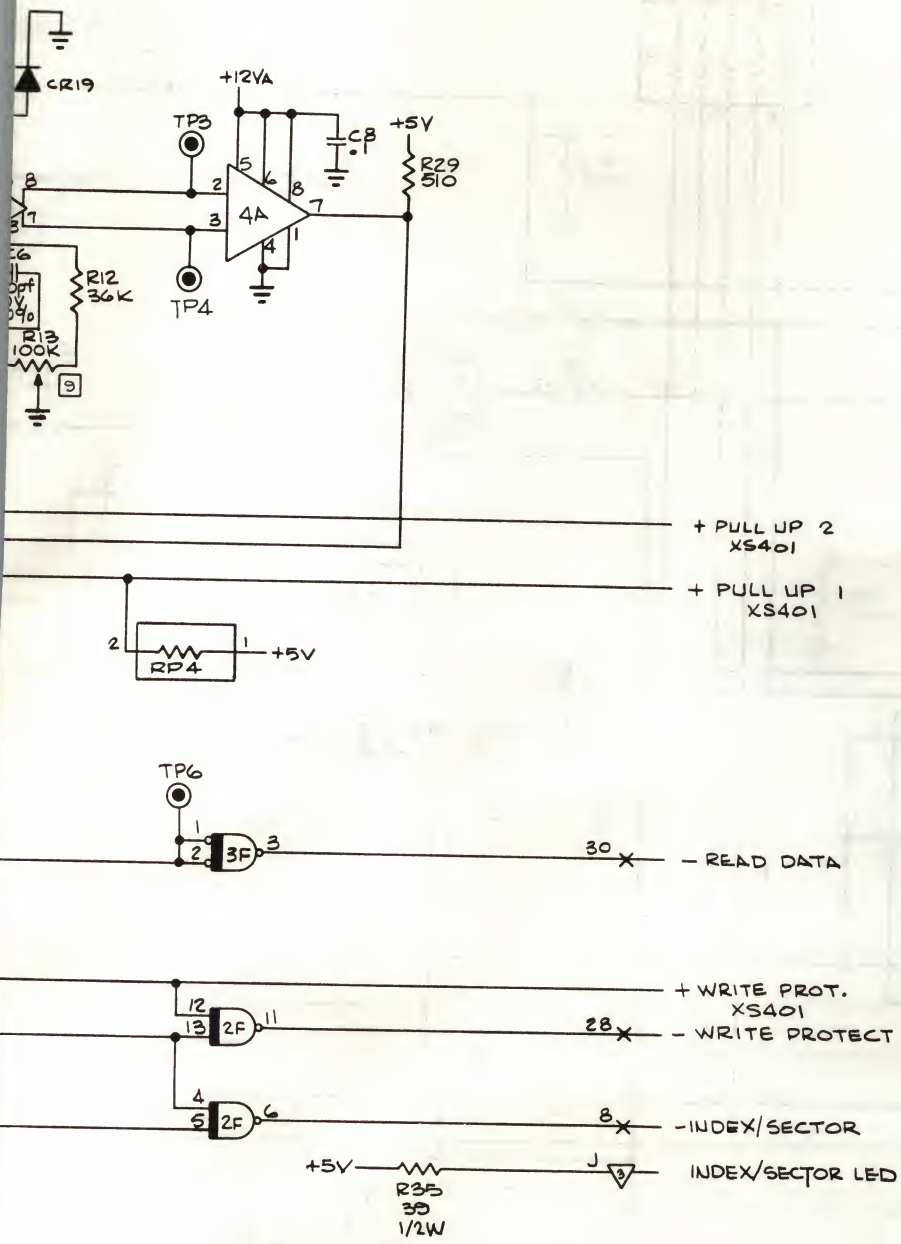
Schematic Diagrams

Control Logic



RESISTOR ARRAY		
VALUE	POSITION	UNUSED
330Ω	RP2, 3	
680Ω	RP1	
150Ω	RP5	
1K	RP4	

TYPE	POS.	UNUSED	VCC (PIN)	GND (PIN)	VDD (PIN)	TYPE	POS.
7400	3E		14	7	—	75453B	3F
7402	3B		14	7	—	LM311N	4A
74LS14	2E		14	7	—	NE592A	3A
7407	2C,4D		14	7	—	7486	4C
7433	2D		14	7	—	9602	4B
7438	2F		14	7	—	2QT2222	2A
7474	2B,1D	1D1 &	14	7	—	2QT2905	1C
74195	3D		16	8	—	150Ω	1E
74LS221	3C		16	8	—	MPQ3725	4F



NOTES: UNLESS OTHERWISE SPECIFIED,

1. ALL CAPACITORS ARE IN MICROFARADS, 50V, +80, -20%.
2. ALL DIODES ARE IN4148.
3. ALL INDUCTORS ARE IN MICROHENRIES, 10%.
4. ALL RESISTORS ARE IN OHMS, 1/4W, 5%.
5. O—O INDICATES SHUNT SELECTABLE OPTION.
6. * INDICATES J1, ▽2 INDICATES J2, ▽3 INDICATES J3, ▽4 INDICATES J4.

[7] COMPONENT NOT INSTALLED.

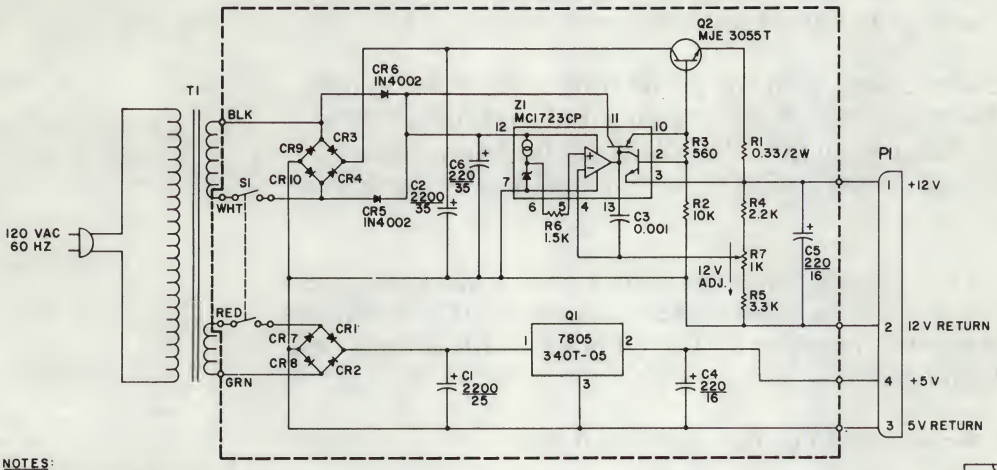
8 PIN 4 OF ID IS GROUND.

[9] R13 VALUE MAY BE 50K.

REF DESIGNATION LAST USED	REF DESIGNATION NOT USED
C31	C20,23,24
CR19	
L5	
Q2	
R42	
RP5	
TP13	

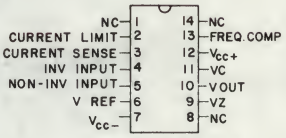
.	UNUSED	VCC (PIN)	GND (PIN)	VDD (PIN)
		8	4	—
		—	1	8
		—	5	10
	4C2	14	7	—
		16	8	—
		—	—	—
		—	—	—
	2	—	—	—
		—	—	—

Power Supply



NOTES:
UNLESS OTHERWISE SPECIFIED:

1. ALL RESISTORS ARE 1/4 WATT, 5%, RESISTOR VALUES IN OHMS, K=1000.
2. CAPACITOR VALUES IN MICROFARADS AND WORKING VOLTAGE.
3. ⟶ INDICATES CLOCKWISE ROTATION.



MCI723CP TOP VIEW

Making a TRSDOS BACKUP

Before you do anything else with your TRSDOS diskette, follow these instructions for making a "safe copy" of your system software. That way, if anything should happen to your original, you won't be "out of business" while you wait to get another one.

Connect the Mini Disk system and power it up as described in the Mini Disk Operation chapter. Be sure your TRSDOS diskette is in Drive 0 when you turn on the CPU. (And just for safety, leave the write protect tape on the TRSDOS diskette until you've duplicated it.)

If you have more than one drives connected, place a blank diskette in drive 1. If not, have the blank diskette handy – BACKUP will tell you when to insert it into drive 0. Do not place a write protect tape on the blank diskette.

After you power on the CPU, the display will read

```
TRSDOS - DISK OPERATING SYSTEM - VER 2.1

DOS READY
-
```

Type:

```
BACKUP ENTER
```

The system will then display:

```
TRSDOS DISK BACKUP UTILITY  VER 2.1
```

If you have only 1 drive connected, type:

```
SOURCE DRIVE NUMBER ? 0 ENTER
DESTINATION DRIVE NUMBER ? 0 ENTER
```

If you have two or more drives, type:

```
SOURCE DRIVE NUMBER ? 0 ENTER
DESTINATION DRIVE NUMBER ? 1 ENTER
```

Now type in the date in MM/DD/YY form. For example, if it's August 3, 1978, type:

```
BACKUP DATE (MM/DD/YY) ? 08/03/78 ENTER
```

TRSDOS will then start the BACKUP procedure. First it will format the blank diskette, locking out any defective tracks; then it will duplicate the contents of the TRSDOS diskette onto it.

If you are using only one drive, BACKUP will tell you when to insert the destination (blank) diskette, and when to re-insert the source (TRSDOS) diskette. During the BACKUP process, you will have to swap the two diskettes several times.

When the process is completed, the message:

```
BACKUP COMPLETE - PRESS ENTER TO CONTINUE
```

will be displayed.

If TRSDOS instead displays the message:

```
BACKUP REJECTED DUE TO (...)
```

then erase the diskette with a bulk eraser (Radio Shack Catalog Number 44-210) and repeat the BACKUP procedure. If it still won't work, you may need to try using another blank diskette.

IMPORTANT NOTICE

The BACKUP utility is provided solely for your personal use in maintaining safe copies of your TRSDOS and data diskettes. BACKUP automatically places copyrighted TRSDOS software onto each destination disk. TRSDOS users may BACKUP the system software solely for personal use.

See the Copyright Notice at the beginning of this Manual for more details.

TRSDOS
An Overview



TRSDOS

Contents of This Section

Introduction 2
Entering a Command 5
File Specification 6

Introduction

TRSDOS, like the entire TRS-80 Microcomputer System, is designed to satisfy a broad range of users, including:

- The novice to computers, who wants to start simply and learn the details gradually
- The experienced programmer, who expects to write complex programs, and may want to use some of the system routines on a machine language level, to accomplish a variety of sophisticated, customized applications
- The pure "user", who is only interested in using programs, not writing them (for example, a clerk using an inventory program on the office TRS-80).

What Is an Operating System?

By the time you finish this book, you'll have a pretty good idea . . . But for the time being, here's an overview.

An operating system is a master program that allows a complex computer system, including various Input/Output (I/O) devices, storage devices and programs, to interact efficiently and with apparent simplicity. The operating system makes sure everything that has to be done, gets done – and you don't even have to know what it is that "has to get done"!

Here's a rather arbitrary breakdown of what an operating system does (see **Glossary** for unfamiliar terms):

- Interfaces the central processing unit (CPU) with the various input/output and storage devices
- Accepts and interprets operator commands
- "Shepherds" your programs (and system utilities you request) in and out of the execution sequence, by allocating CPU time, I/O channels, storage and other system resources
- Handles interrupts, and oversees the execution of both foreground and background tasks
- Provides fundamental routines which would otherwise have to be included in every program; this saves memory and programming time

You don't always have to be aware of the operating system to use it. For example, when you're using DISK BASIC, you don't see TRSDOS at all. But the system is still there, executing a program called BASIC; BASIC, in turn, executes your own programs and commands.

At other times, the operating system may be quite visible to you, allowing you to enter system commands directly. This is the case with TRSDOS and its "DOS READY" mode.

What Is TRSDOS?

The TRS-80 Disk Operating System (TRSDOS) is a comprehensive set of system routines and file management utilities. Much of its complexity (and power) relates to the fact that it is disk-based.

The system is loaded from diskette, and uses diskettes to store internal bookkeeping information as well as data and programs you create. TRSDOS uses completely dynamic disk space allocation, so you can open and manipulate files freely without worrying where they are physically located on the diskette. When a file fills the space currently allocated to it, TRSDOS automatically finds and acquires more space to accommodate additional data (assuming space is available on the diskette).

(All information on a diskette – programs, data, and TRSDOS itself – exists in the form of files. For more information on files, see the **Glossary**, **Files Entry**, and the **Technical Information** chapter.)

In addition to system routines which perform the functions described above under "What is an Operating System?", TRSDOS includes several file management utilities to let you manipulate and modify existing files on the diskette: copy, append, rename, change the protection status, etc.

How TRSDOS uses RAM

TRSDOS consists of:

- an executive program file
- auxiliary system-routine files
- a library-command file
- extended utility files (BACKUP and FORMAT)
- and the DISK BASIC file.

The **executive program** is loaded into RAM on power-up, and remains there at all times while TRSDOS is running. For this reason it is called the "**resident**" TRSDOS program. It includes certain system routines, tables, pointers, and Input/Output drivers.

The **auxiliary system files** contain routines and commands which are loaded as needed to execute your commands and programs. These routines load into an "overlay" area of memory. When TRSDOS has executed the routine, another one may be loaded in the same area, or "overlayed". The use of overlays means that execution of system routines will not affect your memory area (addresses above 51FF hex).

The **library command file** contains the routines for executing most of the operator commands. These routines load into memory addresses from 5200 to 6FFF. Therefore your machine language programs should generally be located above 6FFF. That way they won't be affected by execution of the library commands.

The TRSDOS **extended utility programs** are loaded when you type in their file names, BACKUP and FORMAT. These programs can use all available memory – even the resident TRSDOS program is wiped out when they are loaded.

DISK BASIC is a set of enhancements to LEVEL II BASIC. When you type in its file name, BASIC, it will load into memory beginning at 5200, and begin execution.

Entering a Command

Whenever the prompt,

```
DOS READY
_
```

is displayed, you may enter an operator command. In its simplest form, an operator command is just a single word – a system or library command, the name of an extended utility program, or the name of a user command program. All these categories will be detailed later.

As an example,

```
DIR ENTER
```

tells TRSDOS to display the user file directory for drive 0.

In general, operator commands will require more than one word; for example, to kill (delete) a certain file, you have to specify the file name.

```
KILL XYZ ENTER
```

tells TRSDOS to find the file named XYZ, eliminate it from the directory of the diskette which contains it, and release the space occupied by that file.

In general, an operator command consists of a command followed by one or more file specifications, followed by special parameters:

command [*filespec*] [(*param*)] [TO] [*filespec*] [(*param*)]

where *filespec* is a valid TRSDOS file specification (more below)
param is a parameter which details how the command affects the specified file(s).

If this command format seems complex, don't worry; that's because it's so generalized. The actual commands can be quite simple, as you'll see from the examples given with each command.

Whenever you finish typing in a command, press ENTER . TRSDOS will then process the command as follows:

1) Check to see if it's a system or library command; if so, execute it immediately . . . otherwise

2) Check to see if it's the name of a utility program; if so, execute it via the extended utility package . . . otherwise

3) Examine the diskette directory on each drive to see if the command is listed as a user command file; if so, load and execute the file.

File Specification

A file specification (filespec) is the way you reference a particular file, whether you're operating under TRSDOS, DISK BASIC, or any other command program (e.g., TAPEDISK).

Disk file specifications have the following format:

name[*/ext*] [*.pw*] [*:d*]

where

name is the file name, consisting of from 1 to 8 alphanumeric characters, the first of which must be alphabetic

ext is an optional extension of the name, consisting of from 1 to 3 alphanumeric characters, the first of which must be alphabetic. The extension, if used, must be preceded by a slash symbol.

pw is an optional password, consisting of from 1 to 8 alphanumeric characters, the first of which must be alphabetic. The password, if used, must be preceded by a period symbol.

:d is an optional drive specification, with *d* equal to 0,1,2 or 3, depending on which drive you wish to specify. The drive specification, if used, must be preceded by a colon.

Do not embed blanks in a file specification. If you do, TRSDOS will terminate the filespec at the first blank; if the truncated filespec is valid, you won't receive an error message.

Valid file names:

A	INVNTORY	DATA11
GAMES/BAS	SORTER/VR1	SORTER/VR2
PAYROLL/BAS.SESAME	SECRETS.MYNAME	POETRY/TXT:1
DRIVECHK:1	DRIVECHK:2	AUG3078/DAT.JQD
AUG1578	TAXES/TXT.TEAPARTY:1	CHKWRITR/BAS.VERSION2

To take a completely "filled out" filespec, TAXES/TXT.TEAPARTY:1 refers to a file named TAXES, with an extender TXT, and a password TEAPARTY. This file is referenced to drive 1. If you are creating a file under that filespec, it will be placed on drive 1. If you are reading or writing to the file specified, TRSDOS will reference drive 1 for the file.

What makes a particular filespec unique?

The name, extension and drivespec all figure into the uniqueness of a particular filespec. The password does not.

For example, the following filespecs refer to distinct files:

A	A/BAS	A/CMD	
DRIVECHK:0	DRIVECHK:1	DRIVECHK:2	DRIVECHK:3

However, the following filespecs *cannot* be used to reference distinct files:

RECEIPTS RECEIPTS.AUG3078 RECEIPTS.AUG3178

(There *are* cases where two different passwords are used to access the same file; see **TRSDOS Library Commands,** ATTRIB.)

More on Extensions

The particular extension you use can be purely arbitrary and personalized. Used this way, extensions give you an extra three characters to work with in creating a suitable file name.

Examples:

PAYROLL/AUG PAYROLL/SEP PAYROLL/OCT

However, extensions become more meaningful when they are used as type specifiers, using some convention. Here's a recommended set of extensions:

/BAS	BASIC program file stored in compressed format
/TXT	ASCII text: BASIC program saved in ASCII form, or source file, etc.
/CMD	machine language command file
/CIM	core (RAM) image file, not necessarily executable
/REL	relocatable machine language program file
/SYS	system program – files which are part of TRSDOS. Don't use for your files.
/OV*n*	overlay number *n*
/DVR	I/O driver module

One advantage of this usage is that anyone looking at a directory listing of a diskette will know what kinds of programs he's looking at.

Another advantage is that TRSDOS is equipped to recognize certain extensions. For example, if a file has the extension /CMD, then TRSDOS will load and attempt to execute that file when you type:

filename **ENTER**

omitting the extension /CMD.
That's why you can execute the file BASIC/CMD by typing

BASIC **ENTER**

Similarly, your own programs can be written to recognize extensions.

More on Drive Specifications

If you give a drive specification, TRSDOS will use the specified drive in executing the command. If you omit a drivespec, TRSDOS will search through the directories of all drives in use, starting with drive 0; the first drive with the correct name/extension will be used. However, if the command requires a file creation, TRSDOS will skip over to the first non write-protected diskette.

For example, suppose four files named DRIVECHK are contained on drives 0 through 3. Then every reference to DRIVECHK (no drivespec) would go to drive 0. The filespecs DRIVECHK:0, DRIVECHK:1, DRIVECHK:2, DRIVECHK:3, would allow each of the four files to be accessed.

More on Passwords

The password is assigned when the file is created, and may be changed via the ATTRIB or PROT commands. Files with passwords can only be accessed by reference to the password, or to the diskette's Master Password. So if you assign a password to a file, don't forget it!

It's important to realize that every file has a password, even if you do not specify it explicitly when the file is created. In such cases, a field of 8 blanks becomes the password.

For example, if SAMPLE (a file with no explicit password) exists and you attempt to create a new file, SAMPLE.WATERBOY, TRSDOS will give you a FILE ACCESS DENIED message, since in effect you're trying to access an existing file with the wrong password. The correct password is a string of 8 blanks – which you can omit from the file specification, since 8-blanks is the default password.

TRSDOS Commands



TRSDOS

Contents of This Section

System Commands 2

BASIC2 2
DEBUG 3
TRACE 10

Library Commands 11

AUTO 11
ATTRIB 12
CLOCK 14
COPY 15
DATE 15
DEVICE 16
DIR 16
DUMP 18
KILL 19
FREE 19
LIB 19
LIST 20
LOAD 20
PRINT 21
PROT 21
RENAME 22
TIME 23
VERIFY 24

System Commands

These three commands (BASIC2, DEBUG, TRACE) leave user RAM (hex address 5200-End) "untouched". The necessary code for these commands loads into the overlay area between the resident program and hex 5200. The other commands, referred to as **library commands**, use addresses between hex 5200-6FFF. So locate your machine-language routines above hex 7000 to protect them from the utility commands.

BASIC2 (jump to LEVEL II BASIC)

BASIC2

This command has no arguments or parameters. It simply transfers control to LEVEL II BASIC. Once it has been executed, TRSDOS is no longer resident in RAM. Your TRS-80 will then function as a LEVEL II machine.

You may want to do this to gain memory for programs which don't require disk capabilities. Another possible application would be to LOAD a machine language routine from disk into high memory, and then jump to LEVEL II BASIC via BASIC2, so you can access the routine from LEVEL II, via a USR function.

Example:

```
BASIC2 ENTER
```

```
MEMORY SIZE? ENTER
RADIO SHACK LEVEL II BASIC
READY
>_
```

To re-load TRSDOS, press the Reset button or type

```
SYSTEM ENTER
*? /0 ENTER
```

DEBUG (real-time debugging program)

DEBUG[ϕ(*param*)]

where *param* = ON or OFF, and ON is the default.

DEBUG is a real-time debugging package for use with machine language programs, including both foreground tasks and background programs. (See **Glossary**.) DEBUG lets you examine and alter the contents of the Z-80 registers and RAM locations; jump to specified addresses and begin execution with optional breakpoints; step through programs one instruction (or one CALL) at a time, and more.

All address and byte values in this DEBUG section are given in hexadecimal form – which is the form required by DEBUG.

DEBUG loads into the overlay area; addresses above 51FF are unaffected.

Type:

DEBUG **ENTER**

to enable the debugging facility. Normal TRSDOS command interpretation continues; but the debug program is now set to load and execute under any of the following conditions:

1. When the BREAK key is pressed.
2. After a program is loaded and before its first instruction is executed.
3. Upon detection of a disk-related error.

Note: TRSDOS system routines and execute-only user routines cannot be fully debugged: you can use DEBUG to examine/alter register and RAM contents, but not to single-step, jump, etc., when these protected programs are the "targets" for DEBUG. Furthermore, since DEBUG loads into the overlay area of RAM, you can't use it with other overlay programs and routines.

DEBUG offers two display formats:

register display with indirect RAM
plus any 64-byte "page" of RAM;
full screen, 256-byte page of RAM.

In the register display format, DEBUG displays all the Z-80 registers, organized for interpretation either as two 8-bit registers or as 16-bit register pairs. Since most programs use several sets of register pairs as indirect pointers or indexing registers, 16 bytes of indirect data are presented with each register pair. Each of the flag registers is shown with an ASCII representation of its flag bits.

An additional 64 bytes of memory are displayed in four lines at the bottom of the display.

Here's a typical DEBUG display sequence. Note that the values in your display will typically vary from these.

```
DEBUG ENTER
DOS READY
BREAK
```

```
AF = 3E 28 --1-1---
BC = 0A 3E =>  09 BA C2 60 09 7D 93 C2  60 09 C9 21 27 41 CD D3
DE = 01 04 =>  1A 4D 45 4D 4F 52 59 20  53 49 5A 45 00 52 41 44
HL = 00 54 =>  01 01 5B 1B 0A 1A 08 18  09 19 20 20 0B 78 B1 20
AF'= FF FF SZ1H1PNC
BC'= 4D BE =>  51 51 CD FC 51 7E 23 18  EC 02 02 00 4E 03 32 E7
DE'= 01 07 =>  4D 4F 52 59 20 53 49 5A  45 00 52 41 44 49 4F 20
HL'= 4D 00 =>  F2 51 06 10 CD 65 51 3A  5D 40 FE 41 20 13 CD F2
IX = 40 15 =>  01 E3 03 00 00 00 4B 49  07 58 04 31 3E 20 44 4F
IY = FF FF =>  FF |F3| AF C3 74 06 C3 00  40 C3 00 40 E1 E9 C3 9F
SP = 41 E8 =>  52 04 DD 03 15 40 15 40  18 43 3F 3F 4C 00 E3 05
PC = 00 60 =>  0B 78 B1 20 FB C9 31 00  06 3A EC 37 3C FE 02 D2
     1010 =>  28 10 FE 44 28 0C FE 30  28 F0 FE 2C 28 EC FE 2E
     1020 =>  20 03 2B 36 30 7B E6 10  28 03 2B 36 24 7B E6 04
     1030 =>  C0 2B 70 C9 32 D8 40 21  30 41 36 20 C9 FE 05 E5
     1040 =>  DE 00 17 57 14 CD 01 12  01 00 03 82 FA 57 10 14_
```

In this display, register B contains the hex value 0A, and register C contains 3E. Taking the BC register pair as a pointer, it points to address 0A3E. Therefore, the contents of memory locations 0A3E through 0A4D are shown to the right of the BC = 0A3E =>marker. In this case, address 0A3E contains 09, 0A3F contains BA, etc.

The flag registers F and F′ are handled differently. For these registers, the hex contents of the flag register is displayed, along with a bit-by-bit alphabetic code which makes it easier to interpret the flag status. For example, bit 7 (leftmost bit) is the sign bit, so the alphabetic code shows an S in that position whenever this bit is "set". Here's a complete table of codes for all the flag bits:

bit status	if set	if not set
7 Sign	S	–
6 Zero	Z	–
5 unused	1	–
4 Half-carry	H	–
3 unused	1	–
2 Parity/overflow	P	–
1 Negative	N	–
0 Carry	C	–

In the above display, none of the F flag bits are set (discounting the unused bits 5 and 3), and all of the F′ flag bits are set.

Notice the four additional lines below the PC register display. Each line shows the contents of 16 bytes, starting at the address to the left of the arrow; the four lines always show a total of 64 bytes of contiguous memory i.e., locations with sequential addresses. The starting point in this four-line display is either 0000 or the last command you specified with the D command (more later.)

The blank area in the lower left of the Display is where commands you enter will be displayed.

DEBUG Commands

Note that some commands are executed as soon as you press the specified command key; other commands are executed only when you hit <SPACE> or ENTER , as indicated below.

Command	Entry Required	Operation Performed
A	none	Shows the ASCII or graphics character corresponding to each value displayed. Shows a period when the value is not displayable as an ASCII or graphics character.
C	none	Single-steps next instruction, with CALLS executed in full. (Next instruction is defined by PC register.) Target program cannot be a system or execute-only file.
D*aaaa*	<SPACE>	Sets memory display starting address to *aaaa*. In full screen mode, sets starting address so *aaaa* is contained in display.
G*aaaa*[,*bbbb*[,*cccc*]]	ENTER	Place *aaaa* in PC register and executes with optional breakpoints at *bbbb* and *cccc*.
H	none	Displays all memory and register values in hexadecimal form.
I	none	Single-steps next instruction (defined by PC register). Target program must not be read-protected.
M[*aaaa*]	<SPACE>	Sets the current modification address to *aaaa*. The modification dialog will then be displayed in the lower left of the screen. If *aaaa* is omitted, the last modification address will be used for *aaaa*. If *aaaa* is currently in the display, its contents will be surrounded by a pair of vertical bars.

Command	Entry Required	Operation Performed
R*rp*♭*dddd*	<SPACE>	Loads register pair *rp* with the value *dddd.* *rp* may be any register pair: AF, BC, AF′, BC′, IX, IY, PC, etc.
S	none	Sets display to full screen memory mode, showing 256 contiguous bytes. Press X to return to register display format.
U	none	Dynamic display update mode: lets you observe the execution of a foreground task. Hold down any key for a couple of seconds to exit this mode.
X	none	Sets display to register format; also cancels any command you are in the process of entering, except R-command.
;	none	Increments memory display by one page (in register display mode, page = 64 bytes; page = 256 bytes in full screen mode).
–	none	Decrements memory addresses displayed by one page.

Note: You cannot use the backspace key (←) to delete mistakes made while entering commands. Instead, just hit the X key to cancel the command. Or, if you made the error while typing an address or value, just type the correct address immediately after the incorrect address. DEBUG will only look at the last four digits entered.

For example,

```
D474080  <SPACE>
```

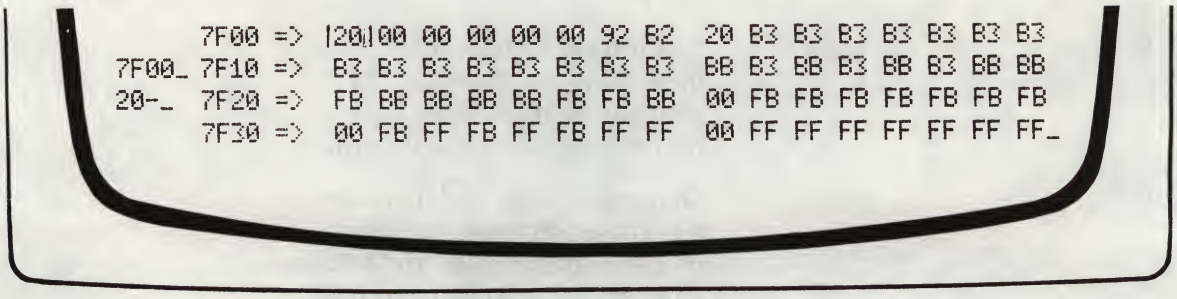
tells DEBUG to display the page of memory containing address 4080.

More on the M-command (modify memory)

Any time you wish to alter the contents of a memory location, type M*aaaa* and press the <SPACE>. This sets the memory modification address to *aaaa* and puts a memory modification prompt in the lower left corner of the Display. For example, typing

M7F00 <SPACE>

produces:



Note the vertical bars around the value of 7F00; These will appear wherever the modification address appears on the screen.

To modify the contents of 7F00, type the new, two-digit contents and press <SPACE>. The display will then be updated, and DEBUG will increment the modification address by one.

To leave an address contents unchanged, simply press <SPACE> without first entering a new contents. This will increment the modification address and leave the previous address unchanged.

To exit the modify memory mode, type X or **ENTER** .

If you simply type:

M <SPACE>

DEBUG will default to the last specified modification address, if any; otherwise 0000 will be used.

Frequently, two values on the display will be highlighted by vertical bars – one in the 64-byte memory display area, and another in the indirect memory area associated with the register pairs.

This is because the contents of the modification address happens to be displayed twice, one directly, one indirectly.

More on the G-command

To return to TRSDOS from DEBUG without re-initializing, type

```
G402D ENTER
```

DEBUG will then be re-entered under any of the three conditions noted above.

To disable DEBUG after using this exit, type

```
DEBUG (OFF) ENTER
DIR ENTER
```

To begin execution at the address in the PC register (while you're in the DEBUG mode), type

```
G ENTER
```

To reinitialize TRSDOS, type

```
G0000 ENTER
```

More on the U-command (update display)

In the Update mode, only foreground tasks are executed. So to see anything happening, you need to look at registers or memory locations used by a foreground task.

The real-time clock makes a good example.

Type:

```
D4040 <SPACE>
```

to display the values 4040 through 4046. These addresses store the time and date, as follows:

address	contents
4040	25mS real-time scheduling counter
4041	seconds
4042	minutes
4043	hours
4044	year
4045	day
4046	month

Now hit U and you'll see the values updated by the clock foreground task.

Other applications for DEBUG

DEBUG can be accessed via DISK BASIC, to help you locate stack pointers, table addresses, etc. See **DISK BASIC**.

DEBUG is also a handy way to create short object code programs, which can then be DUMPed onto diskette.

To disable DEBUG

As long as DEBUG is in the overlay area, TRSDOS may enter the debugging program unexpectedly, for example, upon an error. If you don't want this to happen, disable DEBUG by typing:

```
G402D ENTER          (to return to TRSDOS)
DEBUG (OFF) ENTER
DIR ENTER
```

TRACE (dynamic display of PC register)

TRACE[♭(*param*)]

where *param* = ON or OFF; ON is the default.

The TRACE command enables a foreground task which displays the contents of the user's program instruction counter (PC register) in the upper right of the Video Display. The 4-digit hexadecimal value will be updated every eight milliseconds with the current background program's execution address. For example:

```
TRACE ENTER
```

Since it is a foreground task, TRACE operates at all times – in DOS READY mode, DISK BASIC, or any other program. To temporarily disable TRACE, disable all interrupts (CMD"T" in DISK BASIC). When interrupts are re-enabled CMD"R" in DISK BASIC, TRACE will start up again.

Used with the DEBUG program, TRACE can be invaluable in debugging machine-language programs. It won't be of much use during BASIC program execution, though. To permanently stop TRACE, execute the command:

```
TRACE (OFF) ENTER
```

Library Commands

These commands are overlayed into the RAM area hex 5200-6FFF. They are loaded as requested in blocks; so, for example, DATE and TIME are both loaded when either is requested. TRSDOS will not waste time loading a command if the code is already in RAM.

AUTO (automatic key-in on power-up)

AUTO [/*dos-command*]

where *dos-command* is a filespec for an operator command or an executable command file.

Note: To use AUTO, you must remove the write-protect tab from the system diskette.

The AUTO command lets you modify the power-up sequence, by specifying a command to be executed immediately after power-up.

Typing:

AUTO *dos-command* **ENTER**

causes TRSDOS to write *dos-command* as an "automatic key-in" on the drive 0 diskette, replacing any previous automatic key-ins. From that point on, every time you power up using that TRSDOS diskette, *dos-command* will be keyed in automatically whenever TRSDOS is initialized. An automatic key-in takes the place of keyboard input.

To restore the power-up sequence to normal, type:

AUTO **ENTER**

This will eliminate any automatic key-ins.

Examples:

AUTO CLOCK	on subsequent power-ups, the display clock command will automatically load and execute.
AUTO BASIC	on subsequent power-ups, TRSDOS will load DISK BASIC and begin the initialization dialog.

NOTE: You can override any automatic key-in by holding down the **ENTER** key during power-up. This may be your only way of regaining control of the system, for example, if *dos-command* is not a working command program.

ATTRIB (set protection attributes)

ATTRIB ♭*filespec*♭(*param*[,*param* . . .])

where *param* can be any of the following:

param	meaning
I	make file Invisible to normal Directory command
ACC=*psw1*	assign *psw1* as the new access password
UPD=*psw2*	assign *psw2* as the new update password
PROT=*level*	assign *level* as the new access protection level: (KILL, RENAME, WRITE, READ, EXEC)

The *filespec* must exist on one of the connected drives.

This command lets you alter the protection status of a file, by changing passwords and/or the degree of access granted by a password. (See **TRSDOS Overview**, "File Specifications" section.)

Specifying the I parameter gives the file the invisible attribute. To display Invisible files in the Directory, you have to specify the I parameter in the DIR command. There is no way to remove the I attribute, short of copying the file to a new file which does not have the I attribute.

Example:

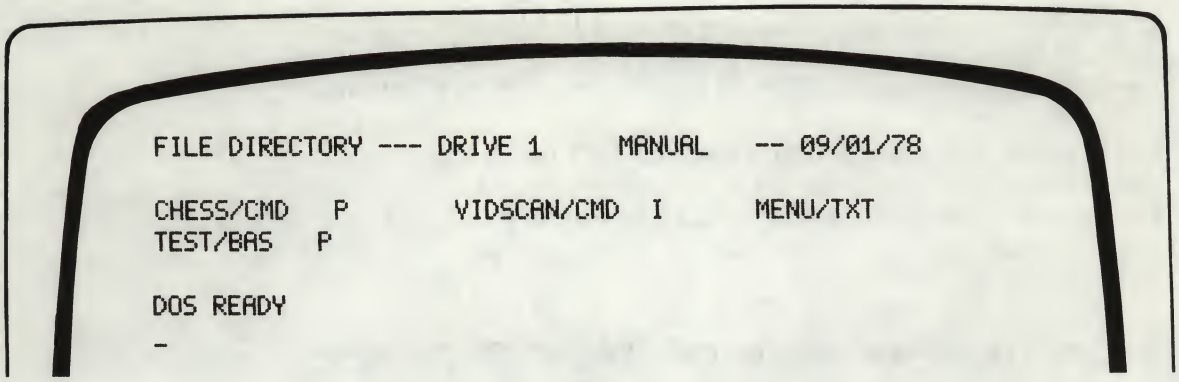
```
ATTRIB VIDSCAN/CMD:1 (I) ENTER

DOS READY
DIR :1 ENTER
```

```
FILE DIRECTORY --- DRIVE 1     MANUAL    -- 09/01/78

CHESS/CMD   P       MENU/TXT             TEST/BAS   P


DOS READY
DIR :1 (I) ENTER
```



All files are protected with two passwords, an access and an update password. Access and update passwords may be identical, and they may consist of all blanks. Use of the **update** password grants total privilege to a file – you can kill, rename, write, etc. Use of the **access** password, on the other hand, grants a limited privilege, as specified by a PROT parameter in the ATTRIB command.

The protection levels form a hierarchy, and each level implies access to all lower levels.

level	privilege
KILL	total privilege
RENAME	rename, write, read, execute
WRITE	write, read, execute
READ	read, execute
EXEC	execute only

When you create a file, the password you specify becomes both the access and the update password. (If you don't specify a password, a string of 8 blanks is assigned as a default password for both access and update.)

Once you have created the file, you can use ATTRIB to assign different values to the access and update passwords. Having two different passwords can be very useful in business applications.

For example, suppose you have a data file, PAYROLL, and you want an employee to use the file in preparing paychecks. Assume the file was created with default (blank) passwords.

Then:

```
ATTRIB PAYROLL (ACC=EMPLOYEE,UPD=MANAGER,PROT=READ)
```

would allow the EMPLOYEE to read the file, while only MANAGER could alter it.

To delete a password (set it to blanks), omit the password after the equals sign in the password specification. For example,

```
ATTRIB PAYROLL.MANAGER (ACC=)
```

sets the access password to blanks, and leaves the update password unchanged.

Note: To access a file from DISK BASIC requires a privilege of READ or higher.

CLOCK (display real-time clock)

CLOCK[ϕ(*param*)]

where *param*=ON or OFF; if no *param* is specified, ON is assumed.

Typing:

CLOCK **ENTER**

causes the internal real-time clock to be forcibly displayed on the top line of the Video Display (PRINT positions 53-60). Any characters present at those locations will be overwritten.

The clock display is updated once a second via a "foreground task". In other words, as long as the interrupts are enabled, TRSDOS will periodically interrupt whatever "background program" is executing (DISK BASIC, TAPEDISK, etc.), and update the clock display.

TRSDOS powers-up in a CLOCK OFF condition.

To stop the display-clock function, execute the command:

```
CLOCK (OFF) ENTER
```

See TIME command for information on the real-time clock.

COPY (make a duplicate file)

COPY␣*filespec1*␣TO␣*filespec2*

Creates a duplicate of *filespec1* under the new name *filespec2*. If filespec2 already exists, its previous contents are lost. The first file (*filespec1*) is unchanged by this command.

You must have at least two disk drives to copy a file from one diskette to another.

Examples:

```
COPY PAGE7/TXT:0 TO PAGE7/TXT:1
```

duplicates PAGE7/TXT on drive 0 onto drive 1, using the same name/extension.

```
COPY OLDFILE/BAS.PDQ TO DEADFILE
```

duplicates OLDFILE under the name DEADFILE. Note that OLDFILE is protected by a password, while DEADFILE is not. DEADFILE will be created on the first non write-protected drive in the sequence 0-3.

DATE (set date)

DATE␣mm/dd/yy

where mm is a 2-digit month specification, mm=01 to 12
dd is a 2-digit day specification, dd=01 to 31
yy is a 2-digit year specification, yy=00 to 99

For example, if it's August 3, 1978, type:

```
DATE 08/03/78 ENTER
```

This command resets the real-time date. At power-on, the date is set to 00/00/00. The date is updated each time the clock cycles through a 24-hour period. The real-time clock calendar includes the logic to account for 28, 29, 30 and 31-day months.

DEVICE

DEVICE

This command has no arguments or parameters. It simply lists all currently defined I/O devices: KI=keyboard, DO=video display, PR=line printer.

DEVICE **ENTER**

DIR (display directory)

Example:

DIR[ϕ:*d*][ϕ(*param*[,*param* . . .])]

where :*d* = a drive specification, *d*=0,1,2 or 3, and 0 is the default
param = any of the following:

param	meaning
S	display all System and non-Invisible files
I	display all Invisible and non-System files
A	display disk space allocation for all files displayed

This command reads and displays the file directory of a specified or assumed drive. If no parameters are specified, only non-Invisible user files will be displayed.

Disk space allocation is indicated as follows: LRL (logical record length), EOF (end of file, i.e., highest record number used), and SIZE (measured in GRANules, where 1 granule = one-half track, or 1.25K bytes).

Examples:

DIR **ENTER**

displays all user files on drive 0. A typical output for this command might be:

```
FILE DIRECTORY --- DRIVE 0     TRSDOS    -- 10/03/78

VIDSCAN2/CMD          CLKAXESS/BAS          SELECTRC/DVR
TBUG/CMD              EDTASM/CMD            GLOSSARY/BAS
LISTER/BAS            TAPEDISK/CMD          KBFIX/CIM
DISKDUMP/BAS          GLOSSACC/BAS          VIDSCAN/CMD

DOS READY
_
```

```
DIR :1 (I,S) ENTER
```

displays all files, including System and Invisible files. A typical output for this command might be:

```
FILE DIRECTORY --- DRIVE 1     MANUAL    -- 09/01/78

BOOT/SYS SIP         DIR/SYS SIP          CHESS/CMD   P
MENU/TXT             TEST/BAS   P

DOS READY
-
```

Note the P beside some files. This indicates they have non-blank passwords.

```
DIR (A) ENTER
```

gives the disk space allocation on drive 0, user files only. Typically:

```
FILE DIRECTORY --- DRIVE 0     TRSDOS    -- 11/18/78

EDTASM/CMD          LRL=  256 / EOF=   27 / SIZE=    6 GRANS
RSM/CMD             LRL=  256 / EOF=   18 / SIZE=    4 GRANS
VHMTBUG/CMD         LRL=  256 / EOF=    0 / SIZE=    2 GRANS
SEQCHECK/TXT        LRL=  256 / EOF=    2 / SIZE=    1 GRANS
TBUG/CMD            LRL=  256 / EOF=    5 / SIZE=    2 GRANS
TAPEDISK/CMD        LRL=  256 / EOF=    2 / SIZE=    1 GRANS
CPRINT/BAS          LRL=  256 / EOF=    1 / SIZE=    1 GRANS
HMRSM/CMD           LRL=  256 / EOF=   18 / SIZE=    4 GRANS


DOS READY
-
```

If a Directory listing cannot fit on the screen, only the first 12 lines will be displayed. Press any key to see the remainder of the listing, in increments of 16 lines.

DUMP (dump memory to disk)

DUMP␣*filespec*␣(START=X'*aaaa*',END=X'*bbbb*'[,TRA=X'*cccc*'])

where *aaaa*, *bbbb*, *cccc* are 4-digit hexadecimal addresses

aaaa = starting point in RAM of the machine language program or data block to be dumped to disk; *aaaa* must be greater than 6FFF.

bbbb = ending point in RAM of the block; *bbbb* must be no smaller than *aaaa*

cccc = transfer address; when TRSDOS attempts to execute the file, it will start at *cccc*. If *cccc* is omitted, 402D will be used. This is the address of the normal re-entry into TRSDOS (i.e., re-entry with DOS READY displayed; no re-initialization).

If *filespec* already exists, its previous contents will be lost.

If *filespec* does not include an extension, TRSDOS will automatically assign the extension CIM (core image) to the file.

Once you have dumped a machine language program onto disk, there are two ways to execute it.

1) Simply type filespec ENTER . TRSDOS will load the file and begin execution at the transfer address.

2) Type DEBUG ENTER and then filespec ENTER After TRSDOS loads the file, it will enter the DEBUG package. PC will contain the transfer address. You can then single step the program (I command), call-step (C command), or execute it in full by typing:
G ENTER

Note: A file with the extension /CMD can be loaded and executed simply by typing the file name, without the extension, and pressing ENTER . TRSDOS will supply /CMD as a default extension.

Examples:

```
DUMP GRAPHICS (START=X'7000',END=X'70A0',TRA=X'7000')

DUMP DATA/CIM:1 (START=X'8000',END=X'8050')
```

KILL (delete a file)

KILL*filespec*

This command deletes the specified file and frees the space for use by the system.

If no drivespec is included in the *filespec*, TRSDOS will search for the first drive which contains *filespec*, and attempt to delete that file. If the diskette is write-protected, TRSDOS cannot KILL the file.

Example:

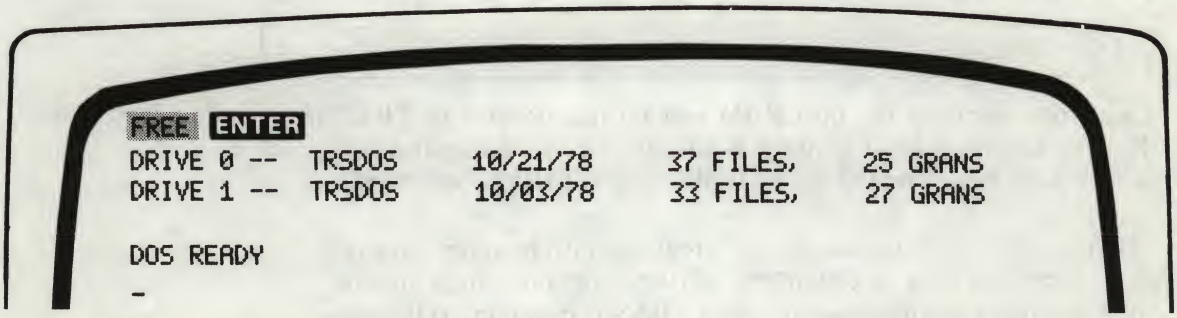
```
KILL OLDFILE/BAS.PASSWORD
```

FREE (display free space on all drives)

FREE

This command has no arguments or parameters. It displays the amount of free space remaining on all drives in use, in terms of files available and unused granules. (Each diskette can contain up to 48 user files; data diskettes have 67 granules available for user files; TRSDOS diskettes, 44 granules.)

For example:



LIB (display library commands)

LIB

Requires no arguments or parameters. This command displays all TRSDOS system library commands available. These are the commands which load between hexadecimal 5200 and 6FFF.

For example:

```
LIB ENTER
```

LIST (list text file contents to display)

LIST␣*filespec*

Reads the specified file and lists its contents on the Video Display. Because LIST gives an ASCII representation of the data in the file, *filespec* should refer to a text file. If you LIST a non-text file, the display will be filled with a meaningless sequence of ASCII and graphics characters.

Text files include:

- BASIC programs saved with the A option
- data files created by BASIC sequential write (PRINT#n) statements
- assembly language source code; etc.

To temporarily freeze the Display during LIST execution, hold down the SHIFT and @ keys until the listing pauses; press any key to resume execution. TRSDOS will only accept such a pause after listing a complete physical record – that's why you need to hold down the SHIFT @ keys until TRSDOS "notices" your pause command.

Example:

```
LIST PROG1/TXT
```

LOAD (load machine language file)

LOAD␣*filespec*

Loads the specified file into RAM and returns control to TRSDOS. The file specified must contain Z-80 object code, and normally would have been created by a DUMP or TAPEDISK command.

LOAD is useful for loading several programs into memory, so that all of them can then be called by a master program, which may be another machine language routine or a BASIC program. (Of course, all the different files must load into non-overlapping areas of RAM.)

To load subsidiary object code programs and then execute them via a master object code program, LOAD each of the subsidiary programs, then type the master filename and press **ENTER** .

Examples:

```
LOAD GRAPHICS
LOAD DATA/CIM:1
```

PRINT (list text file to line printer)

PRINT␣*filespec*

Works just like LIST, only the output is sent to the line printer. The file should be in text (ASCII) form.

Examples:

```
PRINT SEQCHEK/TXT
PRINT PAGE7/TXT:0
```

PROT (use diskette's master password)

PROT[␣:*d*][␣(*param*[,*param* . . .])]

where :*d* = a drivespec, d=0,1,2,3; if no drivespec is given the first drive is used

param can be any of the following:

param	meaning
PW	change Master Password
UNLOCK	remove passwords from all user files
LOCK	assign the master password to all user files

LOCK and UNLOCK are mutually exclusive; use only one.

This command changes the protection status of all non-System files on the specified drive. To use it, you need to know the diskette's Master Password, which is assigned during FORMAT or BACKUP. The diskette you reference must not be write-protected.

Note: Your TRSDOS diskette has the password, PASSWORD.

To change the Master Password, specify PW as a parameter. To remove passwords from all user files, specify UNLOCK. To place the diskette's Master Password on all user files, specify LOCK. (The Master Password then becomes the update and access password for those files.)

Examples:

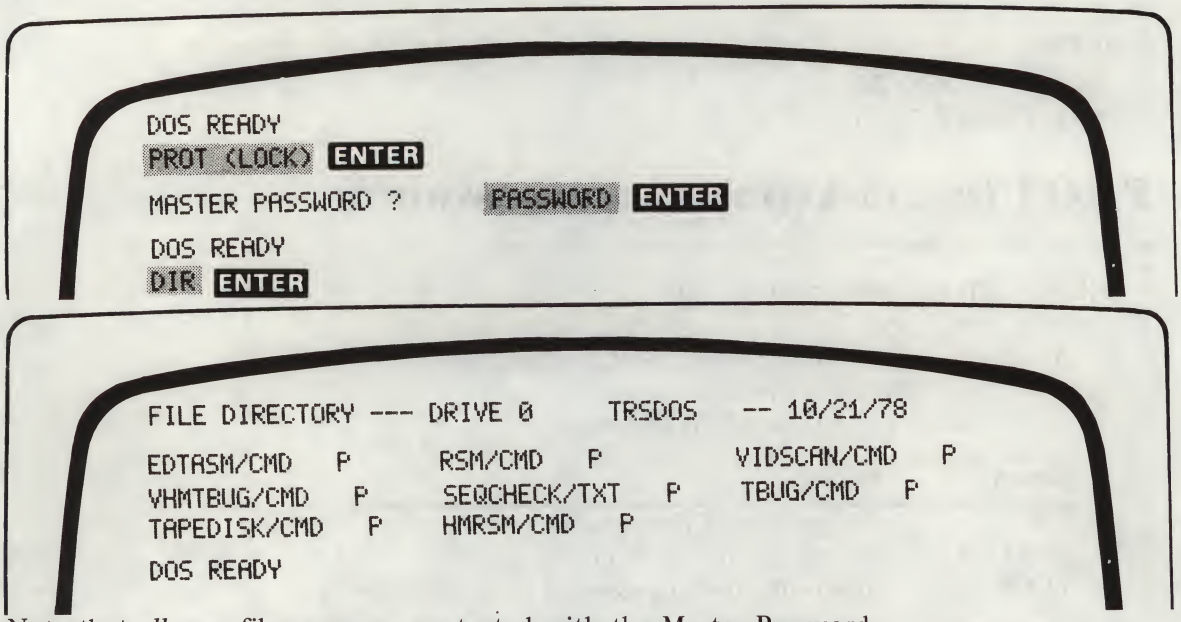
```
PROT :1 (UNLOCK) ENTER
```

After you enter this command, TRSDOS asks for the Master Password for the drive 1 diskette. If you enter the password correctly, TRSDOS will remove all user assigned passwords from files on that diskette.

```
PROT (PW,LOCK)
```

After you specify the Master Password correctly, TRSDOS will prompt you to enter a new Master Password. This new password will be assigned to all user files, since the command included the LOCK option.

A typical display sequence using the PROT command:



Note that *all* user files are now protected with the Master Password.

RENAME

RENAME␣*filename1*[/*ext1*][.*psw*][:*d*]␣TO␣*filename2*[/*ext2*]

where *filename1, filename2* are TRSDOS file names,
ext1,ext2 are extensions
:*d* is a drivespec (d=0,1,2,3)
psw is a password

This command changes a file's name from the first name/extension to the second name/extension. Note that the second name/extension should not include a password or a drivespec. The first file's specification may include a password and drivespec, as required to identify a desired file.

RENAME cannot be used to change a file's protection attributes or to move it to another drive. The previous passwords, protection level, and Directory attributes (Invisible for non-Invisible) will be assigned to the renamed file, and the file will remain on the same diskette.

RENAME also checks to see that the intended new name does not duplicate a filename currently on the same diskette. If it does, the command is cancelled and an error message is displayed.

Examples:

```
RENAME MATHPAK TO MATHPAK/BAS
```
adds an extension to the filename.

```
RENAME ABCDE/DAT TO ABCDEF/DAT
```
changes the file name only.

```
RENAME PAYROLL1/TXT.GSR TO PAYROLL2/TXT
```
changes the filename; the password is retained automatically.

```
RENAME FILE1:3 TO FILE2
```
changes the filename of the file on drive 3 only.

TIME (set real-time clock)

TIME␣*hh*:*mm*:*ss*

where *hh* is a 2-digit hours specification
mm is a 2-digit minutes specification
ss is a 2-digit seconds specification

This command sets the clock. On power-up, the clock is reset to 00:00:00.

Note: TRSDOS maintains a 24-hour/day clock format. After 23:59:59, the clock starts over at 00:00:00, and the day is incremented.

The current time is stored at locations hexadecimal 4040-4046; these values are updated via the realtime clock as long as interrupts are enabled.

Example:

```
TIME 08:24:00
```

See DATE and CLOCK

VERIFY (automatic read-after-write)

VERIFY[ϕ(*param*)]

where *param* = ON or OFF; ON is the default.

VERIFY **ENTER**

causes TRSDOS to verify all user disk writes (for example, file-writes from DISK BASIC). This will be useful when you want to be sure that no data is lost or altered during a disk write. For example, before you COPY a file, you may want to enable VERIFY.

However, when VERIFY is on, disk accesses are only about 50 percent as fast as normal.

Typing:

VERIFY (OFF) **ENTER**

disables the automatic read-after-write verification.
(note that TRSDOS powers up in a VERIFY (OFF) condition.)

Verify does not affect system table and directory writes; they are always verified.

Extended Utilities



TRSDOS

Contents of This Section

TRSDOS Utilities 2
BACKUP 2
FORMAT 4

Auxiliary Utilities 6
TAPEDISK 6
DISKDUMP/BAS 8

TRSDOS Utilities

These are special programs, not strictly a part of TRSDOS, which you can call to perform some very useful functions. Unlike system routines and library commands, these extended programs may use memory locations above hex address 6FFF; therefore any programs you have in RAM may be lost when you load a utility program.

BACKUP (duplicate a diskette)

BACKUP[b:*d1*bTOb:*d2*]

where :*d1* is a specification for the source drive
:*d2* is a specification for the destination drive
d1,*d2* = 0,1,2 or 3.

If you omit the drivespecs, BACKUP will prompt you to enter the source and destination drive numbers one at a time.

This utility duplicates an entire TRSDOS or data diskette. You can use any two drives for the backup, or you can perform the backup using drive 0, by swapping source and destination diskettes when BACKUP tells you to.

If the destination diskette is unformatted, BACKUP will format it, locking out any defective tracks, and will then proceed to copy all source disk files onto it. (If the destination disk cannot contain all the source disk data because of locked out tracks, the backup will be rejected.)

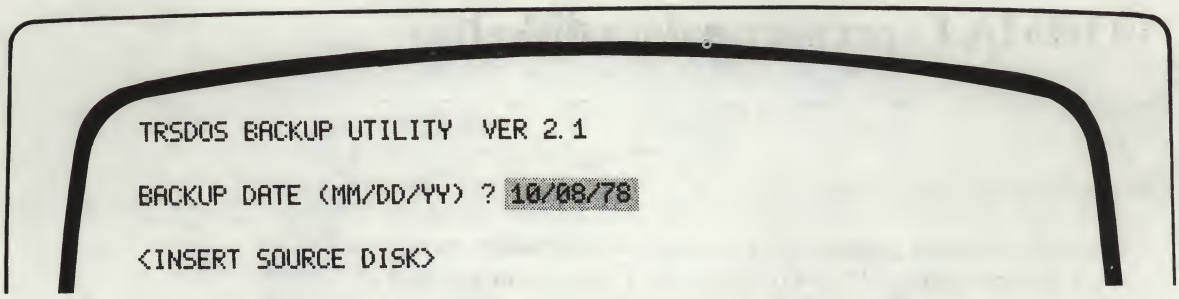
BACKUP will accept a pre-formatted diskette only when its Master Password and Diskette Name match that of the source disk. In this case, BACKUP will skip the formatting step and begin the copy and verify process. If for some reason, BACKUP rejects a diskette, erase the diskette with a bulk eraser and try again.

Examples:

```
BACKUP
BACKUP :0 TO :0
BACKUP :0 TO :1
```

Here's a typical BACKUP sequence, using only Drive 0.

```
DOS READY
BACKUP :0 TO :0 ENTER
```



BACKUP will then prompt you to insert source (original) and destination (duplicate) diskettes as necessary.

When using two drives for the BACKUP, you won't have to do any swapping.

IMPORTANT NOTICE

The BACKUP utility is provided solely for your personal use in maintaining safe copies of your TRSDOS and data diskettes. BACKUP automatically places copyrighted TRSDOS software onto each destination disk. TRSDOS users may BACKUP the system software solely for personal use.

See the Copyright Notice at the beginning of this Manual for more details.

FORMAT (prepare a data diskette)

FORMAT

This utility lets you prepare data diskettes containing a minimum of system information and leaving you with a maximum amount of space for program and data files. (TRSDOS diskettes have 44 granules/55K bytes available for your files; data diskettes, 67 granules/83.75K bytes.

Note: Data diskettes can only be used in drives 1,2, and 3, except during a BACKUP or FORMAT.

FORMAT takes a blank (new or magnetically erased) diskette, records track/sector boundaries on it, then initializes it with directory and bootstrap files. During the formatting process, TRSDOS will let you specify any tracks you'd like to lock out, so you can use them for non-TRSDOS files.

Unless you have another (non-TRSDOS) means of accessing the diskette, don't lock out any tracks.

FORMAT will lock out any defective tracks, to prevent data from being lost in these areas.

If you begin to get READ errors during accesses to a diskette, erase the diskette and re-format it. If there are defective tracks, FORMAT will lock them out, and you'll be left with an otherwise usable diskette.

To lock out tracks...

Specify them individually or as a range.

Example:
1,3-5 locks out tracks 1,3,4,5.

TRSDOS will never try to write to locked-out tracks.

Here is a typical FORMAT sequence, using Drive 1.

```
DOS READY
FORMAT ENTER
```

```
DISK FORMATTER UTILITY  2.1

WHICH DRIVE IS TO BE USED ? 1 ENTER
DISKETTE NAME ? MANUAL ENTER
CREATION DATE (MM/DD/YY) ? 10/08/78 ENTER
MASTER PASSWORD ? TRS80 ENTER
DO YOU WANT TO LOCK OUT ANY TRACKS? Y ENTER
WHICH TRACKS (0-34)? 1,6-8 ENTER
FORMAT THE LOCKED-OUT TRACKS? N ENTER
```

Auxiliary Utilities

TAPEDISK (copy tape file to disk file)

This utility lets you load a SYSTEM tape into RAM, and then dump it into a specified file on the disk. (SYSTEM tapes are created with the Editor/Assembler, TBUG, or supplied by Radio Shack.)

Do not attempt to use TAPEDISK to load tape files which load below hexadecimal address 54F4 (decimal 21748). TAPEDISK uses this area.

Note: Most Radio Shack SYSTEM tapes designed for use with LEVEL II TRS-80's will not work under DISK BASIC, because of differences in RAM usage under DISK BASIC and LEVEL II.

To load and execute TAPEDISK, type:

```
TAPEDISK ENTER
```

TAPEDISK will come up with the prompt,

```
?
```

Any time the prompt is displayed on the current line, you can enter one of the three TAPEDISK commands.

1) Load from tape

```
C
```

is the command to turn on the Recorder. (To use TAPEDISK, you should connect the recorder directly to the TRS-80 tape jack, not to the Expansion Interface jack.)

Type:

```
?C ENTER
```

When the file has loaded, you can load another SYSTEM tape, or enter another command.

2) Dump to disk

F␣*filename*[*/ext*] [*.password*] :*d*␣*aaaa*␣*bbbb*␣*cccc*

where *filename* is a TRSDOS filename
/ext is an optional extension;
.password is an optional password specification;
:d is a required drivespec, *d*=0,1,2 or 3;

aaaa is the hexadecimal starting address in RAM;
bbbb is the hex ending address in RAM;
cccc is the entry point for execution of the file.
All addresses are in 4-digit hexadecimal form.

When you're ready to dump the program from RAM onto disk, type in the F command. For example, if the program loaded into RAM addresses 7000-70FF, and the entry point is at 700A, you'd type:

```
?F USRCODE/CMD:1 7000 70FF 700A ENTER
```

After the dump, the prompt will return.

3) **Exit to TRSDOS**

E

This command returns you to TRSDOS, via the normal re-entry (no re-initialization).

Below is a typical TAPEDISK display sequence.

```
DOS READY

TAPEDISK ENTER

?C

?F GRAPHICS/DAT:0 6A00 6FFF 402D ENTER

?E ENTER

DOS READY
_
```

DISKDUMP/BAS (examine disk file)

This is a BASIC program. To execute it, you must load DISK BASIC first, and then load DISKDUMP/BAS:

```
BASIC ENTER
```

```
HOW MANY FILES? ENTER
MEMORY SIZE? ENTER
RADIO SHACK DISK BASIC VERSION 1.1
READY
RUN"DISKDUMP/BAS" ENTER
```

DISKDUMP lets you look at the contents of any of your disk files. It will help you experiment with various random and sequential disk output statements, and also help you to debug disk I/0 routines.

The program is written to dump to the Line Printer. If you do not have one connected, change all LPRINTs to PRINTs (lines 170,240,250) and change line 160 to:

```
160 GET1,SN
```

This program prompts you to enter the filename and then to enter the sector you want to examine. You can simply press ENTER without a number and the sector-by-sector examination will be sequential, starting with sector 1, the first physical record in the file.

If you specify a sector number higher than the EOF number (end-of file), no error message will be given and the "sector" will appear as zero-value bytes.

The sectors are printed 16 bytes at a time. These 16 bytes are displayed first in hexadecimal code, then with the corresponding ASCII code. The ASCII representation is surrounded by ! symbols. Periods are substituted for bytes which have no alphanumeric representation.

Below is a typical DISKDUMP session.

```
                SECTOR DUMP UTILITY 1.1

FILESPEC: SEQCHECK/TXT

SECTOR NUMBER (OR 'ENTER' FOR NEXT SECTOR): ENTER
```

```
FILESPEC: SEQCHECK/TXT              SECTOR:  1

  0    35 20 43 4C 53 3A 20 43    4C 45 41 52 20 31 30 30   !5 CLS: CLEAR 100!
 16    30 0D 31 30 20 41 24 3D    49 4E 4B 45 59 24 3A 49   !0.10 A$=INKEY$:I!
 32    46 41 24 3D 22 22 54 48    45 4E 31 30 0D 31 35 20   !FA$=""THEN10.15 !
 48    49 46 20 41 24 3D 22 40    22 54 48 45 4E 20 32 35   !IF A$="@"THEN 25!
 64    0D 32 30 20 50 52 49 4E    54 41 24 3B 3A 42 24 3D   !.20 PRINTA$;:B$=!
 80    42 24 2B 41 24 3A 47 4F    54 4F 31 30 0D 32 35 20   !B$+A$:GOTO10.25 !
 96    50 52 49 4E 54 3A 50 52    49 4E 54 22 44 41 54 41   !PRINT:PRINT"DATA!
112    20 49 4D 41 47 45 20 57    49 4C 4C 20 41 53 20 4F   ! IMAGE WILL AS O!
128    4E 20 4E 45 58 54 20 4C    49 4E 45 2E 20 28 22 43   !N NEXT LINE. ("C!
144    48 52 24 28 39 31 29 22    3D 42 59 54 45 20 44 45   !HR$(91)"=BYTE DE!
160    4C 49 4D 49 54 45 52 29    22 0D 33 30 20 46 4F 52   !LIMITER)".30 FOR!
176    49 25 3D 31 20 54 4F 20    4C 45 4E 28 42 24 29 3A   !I%=1 TO LEN(B$):!
192    20 50 52 49 4E 54 20 41    53 43 28 4D 49 44 24 28   ! PRINT ASC(MID$(!
208    42 24 2C 49 25 29 29 43    48 52 24 28 39 31 29 3B   !B$,I%))CHR$(91);!
224    3A 4E 45 58 54 0D 33 35    20 50 52 49 4E 54 0D 35   !:NEXT.35 PRINT.5!
240    30 20 4F 50 45 4E 22 4F    22 2C 31 2C 22 54 45 53   !0 OPEN"O",1,"TES!
```

TRSDOS Technical Information



TRSDOS

Contents of This Section

Memory Organization 2
Disk Organization 2
File Structure 3
System Routines for Assembly I/O 5
    Data/Device Control Blocks 6
    Physical and Logical Records 7
    Fundamental TRSDOS I/O Calls 8
TRSDOS Error Codes/Messages 12

Memory Organization

The TRS-80 Disk Operating System is comprised of 1K of ROM resident CIO (Character-oriented I/O) drivers and 4K of RAM drivers, schedulers, tables, pointers, etc. The ROM resident CIO drivers are also used by LEVEL II BASIC and therefore are part of its 12K ROM requirement.

Since LEVEL II is upward compatible with DISK BASIC, an additional 0.5K of RAM is required for both versions of BASIC. This means that user memory starts at hex 5200, resulting in 11.5K of user RAM in a 16K machine.

Note: The memory which is **completely** untouched by both TRSDOS and DISK BASIC code begins at hex 7000.

TRSDOS is comprised of a resident system and several overlays which are loaded from disk as the need arises (for example, to open or close a file).

The system has a modular design. System entry-point vectors are in the lowest portion of the 4K RAM, followed by the interrupt handling, disk file handling, task scheduling and general purpose resident system routines. System buffers and overlays comprise the last portion of the 4K RAM requirement.

Since all major system commands are actually loaded as needed from disk in the form of utilities (the "library commands" and the extended utility programs), the TRSDOS system facilities can easily be enhanced without affecting the RAM memory requirement.

Disk Organization

Each TRSDOS system diskette contains a TRSDOS system, a utility command library, a file directory, and system tables.

The minimum system overhead amounts to one full track of directory information and a half track of TRSDOS bootstrap program and other information. This means that every TRSDOS diskette is self-loading, although it may or may not actually contain the TRSDOS system. This is done to prevent the Computer from attempting to bootstrap a diskette containing only user data files.

The utility command library is optionally available on the diskette. Since the utility command programs are not always required, it will often be advantageous for multi-drive users to format diskettes for use in drives 1 through 3. Such "data diskettes" contain a minimum of system code, leaving more space for user

files. Maximum file size is limited only by the physical size of the diskette, since a file must be wholly contained on one diskette.

Each diskette is single-sided and has 35 tracks of information. Each track contains 10 sectors of 256 bytes each. See **Mini Disk Operation**, "How a Diskette Works".

Normally, data read/write operations may only be initiated at sector boundaries, and must consist of exactly 256 bytes. However, TRSDOS allows the user to have maximum flexibility with minimal effort by automatically blocking and de-blocking all file accesses to user-specified logical record lengths, even if this requires "spanning" of two sectors.

The system disk file structure allows maximum use of disk file space by automatically segmenting files across a diskette in several small pieces. These pieces are correlated into one logically contiguous file by the system without your needing to know the physical file location. This structure eliminates time-consuming disk-packing operations.

File Structure

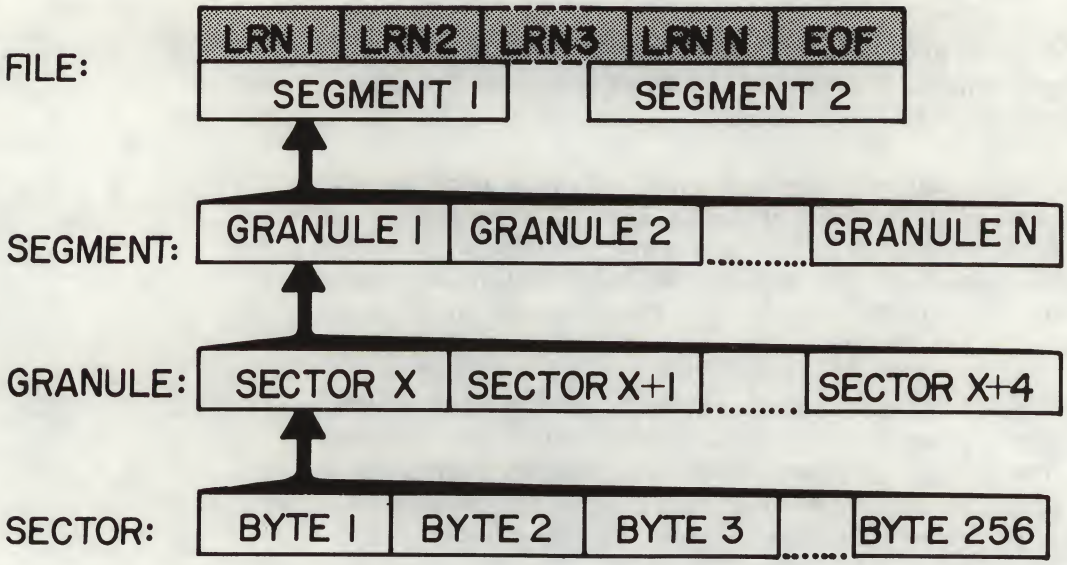
A TRSDOS file is composed of one or more segments of storage space. Each segment consists of from one to 32 physically contiguous granules of storage. A granule is the minimum allocatable unit of storage, and consists of five sectors (1.25K bytes). (See Figure below).

Since a file is always lengthened by granules, a small amount of free storage is generally present at the end of every file. This free storage allows minor file additions to be made in space which is physically contiguous to the file.

The effect is to decrease the amount of "thrashing" present in a file which has had frequent additions made. (A wholly sector-mapped system could not offer this benefit.)

Every time a disk file is extended (either initialized or lengthened), extra granules may be allocated to that file, depending on the file's accumulated length, diskette space, saturation, etc. These extra granules, along with all granules after the one containing the file's EOF mark, are recovered and returned to the system when the file is closed.

A TRSDOS file



LRN: Logical Record Number, used to specify an individual, user-defined logical record. Such a logical record is the smallest unit of information which can be addressed during disk input/output (a physical record is the unit which is actually read from or written to disk).

File: A group of logical records; the largest unit of information which can be addressed by a TRSDOS command.

Sector: A physical record, composed of 256 contiguous bytes.

Granule: The minimum allocatable unit of storage for a particular file.

System Routines for Assembly-Language I/O

This information is provided for customers who wish to write their own assembly level I/O routines. An explanation of the calling sequence and parameters for each necessary I/O routine is given. A knowledge of Z-80 machine code is assumed.

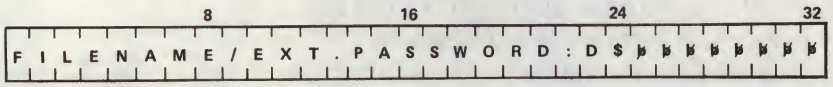
The following notations are standard in this section:

HL=> *xxxx*	Registers HL contain the address of (point to) *xxxx* in machine format. (If address of *xxxx*=34B2H then the values in the registers are: H=34 ; L=B2)
DE=> *xxxx*	Registers DE contain the address of (point to) *xxxx* in machine format. (If address of *xxxx*=5AF1H then the values in the registers are: D=5A ; E=F1)
B= *xx*	Register B contains the numeric value of *xx* in binary form. If *xx*=64 decimal, then B=40H.
A= *xx*	Register A contains the numeric value of *xx* in binary form. If *xx*=127 decimal, then A=7FH. Register A is used to return the TRSDOS error code for I/O calls. A complete list of error codes and their meanings appears at the end of this chapter.
Z=OK	Zero flag is set (OK) if successful return from the system routines.
X'*nnnn*'	Hard RAM address in hex notation (e.g., 402D is X'402D').
LRL	Logical Record Length. 1-255 bytes only. You can define records any length you wish up to 255 bytes maximum. A length of zero is a special case for physical records only, and indicates the LRL=256 bytes.
BUFFER	256 user designated bytes in RAM for TRSDOS to read sectors from or write sectors into. If LRL=0, this area is the responsibility of the user to manage before and after I/O. TRSDOS manages this area if LRL is between 1 and 255 bytes. Do not alter this area when using logical record processing.
UREC	User record: the address of the contiguous RAM byte-string assigned by the user as his logical record area. Its length must be equal to LRL. It is a different area from BUFFER.

DCB before OPEN and after CLOSE:

The DCB is defined as 32 contiguous bytes of RAM designated by the user. Before OPEN and after CLOSE, it is a left justified, compressed (no spaces) ASCII string, as in a standard TRSDOS filespec:

CONTENTS OF 32 - BYTE DCB



Notes: /EXT, .PASSWORD, :D are optional.
$ stands for a carriage return (X'OD')
ß stands for a blank (X'20')

Explanation of DCB while OPEN:

lsb/msb is least significant byte followed by most significant byte in Z80 RAM format (i.e. addr=7CC8 in RAM is C8 7C).

Addr.	Len.	Explanation
DCB+0	3	– Reserved
+3	2	– Physical Buffer address (lsb/msb)
+5	1	– Offset to delimiter at end of current record
+6	1	– File drive number residence
+7	1	– Reserved
+8	1	– EOF offset of last delimiter in last physical record
+9	1	– LRL (logical record length)
+10	2	– NRN (next record no. – open sets=X'0000' – lsb/msb)
+12	2	– ERN (ending record no. – last in file – lsb/msb)
+14	18	– Reserved

NRN Next Record Number defines which record is to be read or written by the next system call for READ or WRITE. It is automatically incremented by one after each system call. In order to process random files, use the POSN call to direct TRSDOS to the record you wish to transfer next.

ERN Ending Record Number is the last record number currently in the file. It is put into the directory at CLOSE time, so if it is expected to be correct, the user *must* close his files after adding records to a file. This value may also be used to position to end of file so that new records may be added to the end of the file. To position to the end of file use a call to POSN with a record number of ERN+1. POSN is described later.

Physical and Logical Records in TRSDOS

A physical record is defined as one sector of disk. One sector of disk contains 256 user data bytes. The artificial term "granule" is defined to be 5 sectors of disk space. There are 2 granules on each of the 35 tracks on the disk. A granule is the least amount of space allocated by TRSDOS. For programming purposes, the physical records in a file are numbered from 0 to N. The largest record number (N) in a file will then be five times the number of granules allocated minus one ((5*G)–1). All TRSDOS granule allocations are made as needed **at the time of write, not when the file is created.**

Bytes	Sectors	Granules	Tracks	Disk
256	1	–	–	–
1280	5	1	–	–
2560	10	2	1	–
89600	350	70	35	1

Disk Space Table : For each 5-1/4" Disk Drive

A logical record is defined by the user of TRSDOS. It may be anywhere from 1 to 255 bytes in length. Once a file is opened with a specific LRL (Logical Record Length), the length is fixed until the file is closed. To change a file's LRL, you must CLOSE it and re-OPEN it with the new LRL.

Each opening of the file sets a single, fixed record-length. TRSDOS will "block" logical records into (or from) one physical record for maximum space utilization on the disk.

Blocking is putting more than one logical record into one physical record. For instance, four 64-byte logical records will fit into one 256-byte physical record. A logical record may be broken into two parts by TRSDOS in order to fill the last portion of one physical record entirely before beginning to use the next physical record (i.e. records are spanned). This occurs when the physical record length is not an even multiple of the logical record length.

If the user wishes to do his own blocking, he may specify a logical record length of 0 bytes at the time of INIT/OPEN and must himself manage the contents of the physical record buffer area of 256 bytes. TRSDOS **will not move** a logical record for the user if LRL=0; in this particular case it will only read/write the physical record to/from the buffer.

Fundamental TRSDOS I/O Calls

There are eight fundamental TRSDOS routines involved in handling file I/O. These are:

INIT	Creates a new file in the directory and opens it. No granule allocation is done.
OPEN	Opens an existing disk file.
POSN	Position for reading/writing a particular logical record.
READ	Reads one logical record into RAM from disk or from the physical buffer.
WRITE	Writes one logical record from RAM onto disk or into the physical buffer.
VERF	Writes then verifies by reading back and comparing to the original data written from RAM. Only pertains to LRL=0 physical records.
CLOSE	Closes an open file.
KILL	Closes a file and erases it from the directory.

The detailed calling sequences and discussions for each of these routines follow. Note that **all** of these system calls use register F and do not restore its value before return. In order to properly apply this data, you should read through all of these descriptions and clear up all of the points that are not obvious to you by using other reference materials. If you are successful in doing this you will find that TRSDOS is a workable tool for your programming ideas. The jump vectors supplied here and the descriptions especially pertain to TRSDOS Version 2.1 only. Future releases of TRSDOS may alter some of these descriptions or addresses.

INIT (jump vector = X'4420')

INIT is provided as an entry point to TRSDOS which will create a new file entry in the directory and open the DCB for this file. INIT scans the directory for the filespec name given in the DCB. If the filespec name is found, INIT simply opens the file for use. If the name is not found, a new file is created with the filespec name.

entry: HL= >BUFFER (see beginning of this section for notation)
DE= >DCB
B= LRL
CALL 4420H

exit: Z=OK
C carry flag is ON if a new file was created
A=TRSDOS error code. (Error codes listed at end of this chapter)

OPEN (jump vector = X'4424')

OPEN provides a way to open the DCB of a file which already exists in the directory. The DCB **must** contain the filespec of the file to be opened **before** entry to OPEN.

entry: HL=> BUFFER
DE=> DCB
B= LRL
CALL 4424H
exit: Z=OK
Z=0 if file does not exist.
A=TRSDOS error code.

POSN (jump vector = X'4442')

POSN positions a file to read or write a randomly selected logical record. Since it deals with logical records, the proper computation is done to locate which physical record(s) contain the data. Following a POSN with a READ or WRITE will transfer the record to/from RAM.

Note that positioning to logical record zero sets the file to read the first logical record in the file. To position to end of file in order to add new records onto the end, use the record number ERN+1 (see page 2).

entry: DE=> DCB (must have been opened previously)
BC= Logical record number to position for.
CALL 4442H
exit: Z=OK
A=TRSDOS error code.

READ (jump vector = X'4436')

If LRL>0, READ transfers the logical record whose number is in the DCB as NRN (see page 2) into the RAM area addressed as UREC for the length LRL as defined at open time. The record comes from the RAM BUFFER defined at open time. If TRSDOS must read a new physical record to satisfy the request, it will do so. "Spanned" logical records will be re-assembled as necessary. READ automatically increments NRN by one in the DCB after the transfer is completed. INIT/OPEN sets NRN=X'0000' in order to read the first record with the first READ.

If LRL=0, READ transfers one physical record into the RAM BUFFER, which was defined at open time, from the disk file. Registers HL are ignored. READ increments NRN as above.

entry: HL= > UREC if LRL is not zero. Unused if LRL=0.
DE= > DCB
CALL 44364
exit: Z=OK
A=TRSDOS error code. (EOF=X'1C' or X'1D')
(see errors 28,29 for EOF or NRF)

WRITE (jump vector = X'4439')

IF LRL>0, WRITE transfers the one logical record from the RAM area addressed as UREC for the length LRL as defined at open time. The record goes into the RAM BUFFER which was defined at open time. If TRSDOS must write a physical record in order to satisfy the request, it will do so. "Spanning" will be handled by TRSDOS as necessary. At INIT/OPEN time the DCB value of NRN is set to X'0000' so that the first record will be written. After each logical record is transferred, the NRN value in the DCB will be incremented by one.

IF LRL=0, WRITE transfers one physical record from the RAM BUFFER into the disk file using the NRN in the DCB. BUFFER IS DEFINED at INIT/OPEN time only. The DCB value NRN is updated as above, after the WRITE.

entry: HL= > UREC if LRL is not zero. Unused if LRL= 0
DE= > DCB
CALL 4439H
exit: Z=OK
A=TRSDOS error code.

VERF (jump vector = X'443C')

The only difference between VERF and WRITE is that VERF writes one physical record to disk and then reads it back into a special TRSDOS RAM area not defined by the user. This special area and the original write buffer are then compared byte by byte to assure that the record was successfully written.

entry: HL= > Same as WRITE above.
DE= > DCB
CALL 443CH
exit: Z=OK
A=TRSDOS error code.

CLOSE (jump vector = X'4428')

CLOSE closes a file from the last processing done. **It is very important to do a CLOSE on every file opened before the program ends.** If you do not close a file, the directory entry for this file is incorrect if any new records have been written into the file. Other cases are not given here, but it is very important to TRSDOS that all of the "housekeeping" is complete for file management.

entry: DE=> DCB
CALL 4428H
exit: Z=OK
A=TRSDOS error code.

KILL (jump vector = X'442C')

KILL deletes the directory entry for an open file and then completes the close on the DCB. The disk space released by the old file is now re-useable for other purposes. Otherwise KILL is the same as CLOSE.

entry: DE=> DCB
CALL 442CH
exit: Z=OK
A=TRSDOS error code.

Supplementary Information

Other routines and addresses which may be of interest are defined here. Pay particular attention to the error routine. It does NOT perform error recovery. It displays TRSDOS error messages on the video display.

(1) CALL 402DH – Normal return to TRSDOS at program end.

(2) X'4318': address of the 64-byte buffer that contains the last TRSDOS command that was entered. Useful to decode special parameters entered when program was executed (run).

(3) If HL => 8-byte buffer, then:
CALL 446DH returns the time of day into the 8 bytes in the ASCII format – HH:MM:SS
CALL 4470H returns the date into the 8 bytes in the ASCII format – MM/DD/YY

Binary forms of the time and date are located in TRSDOS RAM at these locations:
X'4040' clock – real time clock heartbeat count. 25ms.
X'4041' time – binary – 3 bytes – sec,min,hrs
X'4044' date – binary – 3 bytes – yr, day, mon

(4) Printing TRSDOS error codes on the video display.

```
        CALL  4420H     Example of system I/O call.  Any call
                        is ok.  Zero flag not set means an error
                        has occurred during the I/O attempt.
        JR    Z,OKGO    Ignore error message display if no
                        error.
        OR    80H       Optional for detailed error message.
                        Register A already contains proper
                        code for a single line message display.
        CALL  4409H     Display error message on video screen.

              Optional user error recovery code goes here

        OKGO continue with program here - - -
```

TRSDOS Error Codes – Returned in Register A

decimal number	prob. causes*	error description
00	–	No error
01	MD	Parity error during header read
02	D	Seek error during read
03	XK	Lost data during read
04	MD	Parity error during read
05	FMD	Data record not found during read
06	P	Attempted to read system data record
07	P	Attempted to read system data record
08	UP	Device not available
09	MD	Parity error during header write
10	D	Seek error during write
11	XC	Lost data during write
12	MD	Parity error during write
13	FMD	Data record not found during write
14	XD	Write fault on disk drive
15	UDX	Write protected diskette
16	PS	Illegal logical file number (dcb bad)
17	MPDS	Directory read error
18	MPDS	Directory write error
19	UP	Illegal file name (dcb bad)
20	MPDS	GAT read error (Granule Allocation Table)
21	MPDS	GAT write error
22	MPDS	HIT read error (Hash Index Table)
23	MPDS	HIT write error
24	UP	File not in directory
25	UP	File access denied (protection violation)

*See Explanation, next page.

decimal number	prob. causes	error description
26	UP	Directory space full (48 files max)
27	UP	Disk space full (70 granules max)
28	P	EOF encountered (End Of File)
29	P	NRF (No Record Found) out of file range
30	UP	Full directory. File can't be extended.
31	UP	Program not found
32	UP	Illegal drive number specified
33	UP	No device space available for new device
34	MPUS	Load file format error. Not a program.
35	XCS	Memory fault
36	PUXC	Attempted to load ROM memory
37	P	Illegal access attempted to protected file
38	UP	File has not been opened
39-62		Not defined yet. Reserved
63	P	Unknown error code

Explanation of probable cause codes: (column 2)

C = TRS80 CPU fault
D = Disk drive fault
F = Diskette not formatted
M = Diskette media fault
P = User program error
S = TRSDOS fault. Reboot
U = User procedural error
X = Expansion Interface fault

DISK BASIC



LANGUAGES

Contents of This Section

Introduction 2
Enhancements to LEVEL II 5
Disk Features 26
File Manipulation 28
File Access 33
Sequential Access Techniques 60
Random Access Techniques 65
DISK BASIC Error Messages 77

Introduction

DISK BASIC is a set of enhancements to LEVEL II BASIC, plus features to allow disk input/output of BASIC programs and data. It is a memory image file stored on the TRSDOS software diskette with the name BASIC and extension /CMD.

When DISK BASIC is loaded into RAM, it automatically takes control of the LEVEL II BASIC ROM program, using almost all of its routines and adding others. This is possible because LEVEL II was designed with upward compatibility built-in. (You can see this by examining the memory map for LEVEL II, in particular, hex addresses 37DE-37EC.)

When loaded, DISK BASIC occupies approximately 5.8K bytes of RAM, beginning at hex address 5200 (decimal 20992).

To load and execute DISK BASIC, first power-up the Disk Operating System (see System Operation), so that

```
DOS READY
```

is displayed. Now type:

```
BASIC ENTER
```

TRSDOS will load BASIC into RAM, and BASIC will begin the "initialization dialog". This is a series of questions and answers which tell BASIC how to organize memory according to your needs.

The first question is,

```
HOW MANY FILES?_
```

You repond with the maximum number of disk files that will be open (in use) at any one time – any number from zero to 15.

(Every program or data set you store on the disk is referred to as a "file". In fact, everything on the disk, including system software, exists in the form of files.)

The number you enter tells BASIC how many disk I/O **buffers** and **data control blocks** to create (for definitions, see **Glossary**). If *n* files are to be in use at once, then *n* buffers will be required. Each buffer will take 290 bytes from your available RAM (256 for the buffer plus 34 for a data control block [DCB]), so don't enter an unnecessarily large number.

If you simply press ENTER without entering a number, BASIC will use a default value of 3; so you'll be able to have 3 file buffers in use at once.

Note: DISK BASIC automatically creates a buffer for loading, saving and merging BASIC programs. This buffer exists in RAM below any data file buffers you may request. It is always available for program I/O, regardless of how you answer the FILES? question.

Suppose you're going to be using 2 files: 1 for inputting data, 1 for outputting data. Then you might answer 2 to the FILES? question. However, if only 1 of these files will be open at once, then you really only need to reserve 1 file buffer/control block.

Examples:

```
HOW MANY FILES? 0 ENTER
```

causes BASIC to set aside zero buffers for I/O to disk files. You won't be able to open files, but you will have the maximum amount of RAM for use by your program.

```
HOW MANY FILES? 15 ENTER
```

tells BASIC to create 15 I/O buffers and control blocks; you will then be able to have 15 files open at once; however, this will reduce your available memory by 15*290 = 4350 bytes.

```
HOW MANY FILES? ENTER
```

tells BASIC to use a default of 3 for the number of files to be in use at once.

After you answer the FILES question, BASIC will ask:

```
MEMORY SIZE?_
```

You respond with the highest memory address (in decimal form) you want BASIC to use for storing and executing your BASIC programs. Addresses above the number you specify will then be protected from use by BASIC.

Here's why you might want to protect memory:
You can load machine-language programs or routines into high memory, and then access these routines from DISK BASIC via specially defined USR*n* functions, or via the SYSTEM command. These machine language routines may be loaded from tape using the SYSTEM command, LOADed in the DOS READY mode, or placed in memory one byte at a time using either DEBUG or BASIC POKE commands. If you do not reserve memory, such routines will be destroyed during execution of BASIC statements.

Example:

```
MEMORY SIZE? 32000 ENTER
```

causes BASIC to protect addresses above 32000. If you have 16K of RAM, this means that you'll have 32767-32000= 767 bytes protected for storing your machine language routines.

If you don't want to reserve any memory, just press **ENTER** without typing a number.

```
MEMORY SIZE? ENTER
```

You will then have the maximum amount of RAM available for use by BASIC.

Refer to the Memory Map for decimal addresses of the various TRS-80 memory configurations (16K, 32K, 48K).

After you answer the MEMORY SIZE question,

```
RADIO SHACK DISK BASIC VERSION 1.1
READY
>_
```

will be displayed. You are now operating under DISK BASIC.

To exit BASIC and return to the DOS READY mode, type:

```
CMD"S" ENTER
```

This results in a normal return to DOS – without re-initialization of the system. If you have a BASIC program in RAM, it will be lost, so be sure to save it on disk or tape before using CMD"S".

Enhancements to LEVEL II BASIC

DISK BASIC adds many features to LEVEL II which are not disk-related. They are listed below along with abbreviated descriptions. Detailed descriptions follow in alphabetical order.

&H	Hexadecimal-constant prefix
&O	Octal-constant prefix
CMD"D"	Enable and load the real-time debugging program
CMD"R"	Enable interrupts (start real-time clock)
CMD"S"	Normal return to TRSDOS (jump to EXIT routine)
CMD"T"	Disable interrupts (turn off real-time clock)
DEF FN	Define an implicit BASIC-statement function
DEF USR	Define the entry point for an external machine-language routine
INSTR	Instring function; find substring in target string
LINE INPUT	Input a line from keyboard
MID$=	Replace portion of target string (used on left of equals sign)
TIME$	Get time and date from real-time clock
USR*n*	Call external routine (*n*=0,1,2,...,9)

Cassette Operations

Before any BASIC cassette input or output operation, you must disable interrupts with the CMD"T" command. This is because such cassette operations are timing-sensitive and cannot work if they are being interrupted every 25 milliseconds. When the cassette operation is complete, you can re-enable interrupts by executing the statement CMD"R".

CLOAD allows no filename in DISK BASIC. Therefore you cannot use such a filename to sort through several tape files. CLOAD will always load the first file encountered on the tape. CSAVE, however, still requires the filename. This way, programs CSAVEd under DISK BASIC can be loaded and sorted through via the LEVEL II CLOAD"*filename*" command.

CLOAD? (CLOAD-verify), used in LEVEL II to compare a BASIC program in RAM with one on tape, will not work with programs saved on tape under LEVEL II. **It will work with programs saved under DISK BASIC.**

Error Messages

When an error occurs, DISK BASIC "spells out" the full error message, not just the abbreviation. This saves you from having to look it up.

Example:

```
ERROR(14) ENTER
```

DISK BASIC responds with:

```
OUT OF STRING SPACE
```

Note: The ERROR function, used to simulate error conditions, will work only with non-disk error codes.

&H and &O (hex and octal constants)

Often it is convenient to use hex (base 16) or octal (base 8) constants rather than their decimal counterparts. For example, memory addresses and byte values are easier to manipulate in hex form. &H and &O let you introduce such constants into your program.

&H and &O are used as prefixes for the numerals that immediately follow them:

&H*dddd*	
where *dddd*	is a 1 to 4 digit sequence composed of hexadecimal numerals 0,1,...9,A,B,...,F.
&O*ddddd*	
where *ddddd*	is a sequence of octal numerals 0,1,...,7. and &O*ddddd*< =177777 decimal. **Note:** The O can be omitted from the prefix &O. Therefore &O*ddddd*=&*ddddd*.

The constants always represent signed integers.
Therefore any hex number greater than &H7FFF, or any octal number greater than &O77777, will be interpreted as a negative quantity. The following table illustrates this:

Octal	Hex	Decimal
&1	&H1	1
&2	&H2	2
&77777	&H7FFF	32767
&100000	&H8000	–32768
&100001	&H8001	–32767
&100002	&H8002	–32766
&177776	&HFFFE	–2
&177777	&HFFFF	–1

Hex and octal constants cannot be typed in as responses to an INPUT prompt or be contained in a DATA statement. Often the hex or octal constant must be enclosed in parentheses to prevent a syntax error from occurring.

Examples:

```
PRINT &H5200,&O51000
```

prints the decimal equivalent of the two constants (both equal 20992).

```
POKE &H3C00,42
```

puts decimal 42 (ASCII code for an asterisk) into video memory address hex 3C00.

```
100 FOR I=(&H3C00) TO (&H3FFF) STEP (&H40)
200 IF A=(&H37E8) THEN A=A+1
300 POKE A%, (X% AND &HFF)
```

Masks the most significant byte of X% and POKEs the result into location A%.

CMD"D" (execute DEBUG program)

CMD"D"

Executing this statement causes the TRSDOS debugging program to load and execute. (See TRSDOS Commands, DEBUG.) Your BASIC program is unaffected, since DEBUG loads below DISK BASIC.

To return to BASIC without re-initialization, type

G **ENTER**

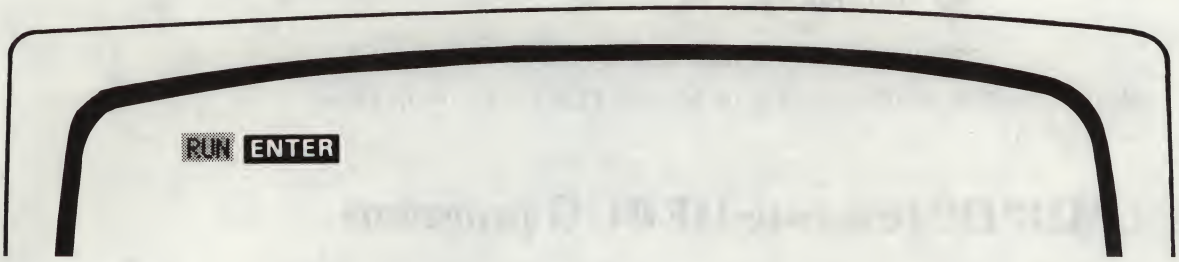
The READY message will appear and you can continue in BASIC.

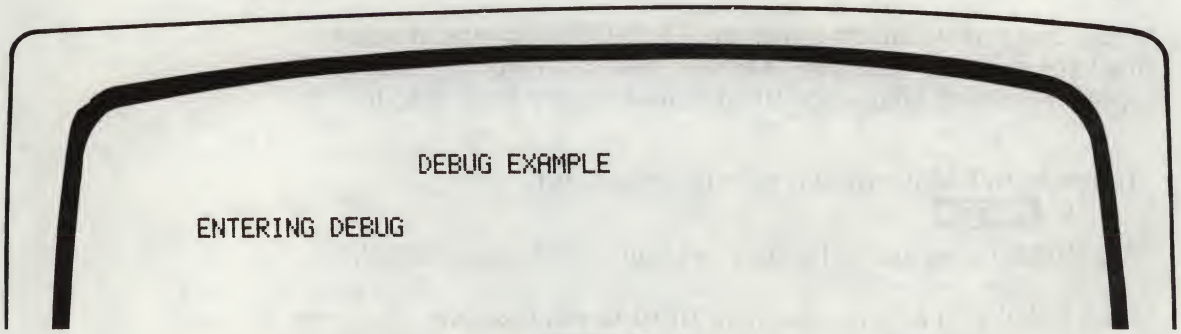
Once CMD"D" has been executed, DEBUG will take over whenever you press the BREAK key. Pressing G **ENTER** will return you to BASIC again. Type CONT to continue any program that was executing when you typed BREAK.

To return from DEBUG to the BASIC initialization sequence, type G5200 **ENTER** . You will lose any BASIC program text or variable values.

Examples:

```
100 '         PROGRAM: DEBUG
110 ' EXAMPLE OF EXECUTION WITH DEBUG WITHIN A PROGRAM
120 '
130 CLS: PRINT TAB(15); "DEBUG EXAMPLE": PRINT
140 PRINT"ENTERING DEBUG"
150 FOR I=1 TO 500: NEXT I   'DELAY A WHILE
160 '
170 ' *** ENTER DEBUGGING PACKAGE ***
180 '
190 CMD"D"
200 '
210 ' *** RETURN HERE WHEN "G" ENTER TYPED IN DEBUG ***
220 '
230 CLS: PRINT: PRINT "YOU HAVE RETURNED FROM DEBUG"
240 END
```





```
AF = 44 42 -Z----N-
BC = 69 01 =>  57 49 54 48 49 4E 20 41  20 50 52 4F 47 52 41 4D
DE = 69 B8 =>  44 22 00 C3 69 C8 00 3A  93 FB 00 FD 69 D2 00 3A
HL = 40 B7 =>  69 55 FF FF FF FF FF FF  FF 00 00 00 00 00 00 54
AF'= FF FF SZ1H1PNC
BC'= 4D BE =>  51 51 CD FC 51 7E 23 18  EC 02 02 00 4E 03 32 E7
DE'= 01 07 =>  4D 4F 52 59 20 53 49 5A  45 00 52 41 44 49 4F 20
HL'= 4D 00 =>  F2 51 06 10 CD 65 51 3A  5D 40 FE 41 20 13 CD F2
IX = 40 15 =>  01 E3 03 00 00 00 4B 49  07 58 04 31 3E 00 44 4F
IY = FF FF =>  FF F3 AF C3 74 06 C3 00  40 C3 00 40 E1 E9 C3 9F
SP = BD 6C =>  BA 69 1E 1D 00 00 04 04  20 00 00 00 00 00 00 00
PC = 57 08 =>  E1 C9 3A 29 5B F6 C0 CD  09 44 E1 C9 D7 E5 3E 11
      1010 =>  28 10 FE 44 28 0C FE 30  28 F0 FE 2C 28 EC FE 2E
      1020 =>  20 03 2B 36 30 7B E6 10  28 03 2B 36 24 7B E6 04
      1030 =>  C0 2B 70 C9 32 D8 40 21  30 41 36 20 C9 FE 05 E5
G ENTER 1040 =>  DE 00 17 57 14 CD 01 12  01 00 03 82 FA 57 10 14
```

```
YOU HAVE RETURNED FROM DEBUG
READY
>_
```

CMD"R" (start clock [enable interrupts])

CMD"R"

Execute this command immediately after completion of a cassette input/output operation to re-start the real-time clock. See CMD"T".

CMD"S" (return to TRSDOS)

CMD"S"

Execute this command to initiate a normal return to the Operating System command mode. This will not re-initialize the system, but merely get you out of BASIC.

Be sure to save any BASIC program on disk or tape before using CMD"S", as your resident BASIC program will be lost.

CMD"T" (stop clock [disable interrupts])

CMD"T"

You must execute this command immediately before any BASIC tape input/output operation. Such operations are timing sensitive and cannot allow the interrupt-driven tasks (such as the real-time clock, TRACE, and CLOCK-display) to "steal" time.

Here are the commands which must be preceded by execution of CMD"T":

CLOAD	CLOAD?
INPUT#-1	CSAVE
INPUT#-2	PRINT#-1
SYSTEM	PRINT#-2

After completion of these operations, you can execute a CMD"R" to re-enable interrupts.

Example:

```
10 OPEN"I",1,"TEST/BAS"
20 CMD"T": INPUT#1,A,B,C
30 CMD"R"
```

Note: After CMD"D", you can use CMD"T" to prevent BASIC from transferring control to the DEBUG program when BREAK is pressed.

DEF FN (define function)

DEF FN *var1*(*var2*[,*var*...])=*exp*

where *var1* will be the name of the function, and is any valid LEVEL II variable name
var2 and subsequent var-items are used in defining what the function does
exp is an expression usually involving the variable(s) passed on the left of the equals sign

This statement lets you create your own implicit functions. That is, you only have to call it by name and the implicit function you defined will automatically be performed. Once a function has been defined with the DEF FN statement, you can call it simply by referencing the function name prefixed by FN. You can use it exactly as you'd use one of the intrinsic functions, e.g., SIN, ABS, STRING$.

The type of variable used to name the function determines what type of value the function will return. For example, if the function name has the single-precision attribute, then that function will return a single-precision value – regardless of the precision of the arguments.

Examples:

```
DEFFNA$(TITLE$,GRAPHICS%)=STRING$(LEN(TITLE$),GRAPHICS%)
```

The function A$ will require two arguments – one integer, one string; and it will return a string value.

```
DEFFNRC!(A)=1/(A*A)
```

The function RC! requires one argument, and returns a single-precision value, regardless of the precision of the argument.

The particular variable names you use as arguments in the DEF FN statement are not assigned to the function; when you call the function later, any valid variable name of the same type can be used. Furthermore, using a variable as an argument in a DEF FN statement has no effect on the value of that variable.

The function must be defined with at least one argument – even if this argument is not actually used to pass a value to the function. For example:

```
DEF FNR(A)=RND(0)
```

Examples:

```
10 DEFFNMLT(A,B)=A*B
20 INPUT "ENTER ARGUMENTS";X,Y
30 PRINT "PRODUCT IS"; FNMLT(X,Y)
```

Notice that FNMLT is defined with arguments A,B, but that when the function is called in line 30, variables X and Y are used. Any two valid variable names can be used to pass values to the function.

`DEF FNRR(A,B)=A+INT(B*RND(0))`	Returns a random number between A and B.
`DEF FNL8$(A$)=LEFT$(A$,8)`	Returns first 8 characters of string argument
`DEF FNX#(A#,B#)=(A#-B#)*(A#-B#)`	Returns double-precision value of "the square of the difference"

```
100 '          PROGRAM: STRING
110 ' EXAMPLE OF A STRING DEFFN FUNCTION
120 '
130 ' ******* FUNCTION TO CONCATENATE STRINGS *******
135 '
140 DEF FNADD$ (A$, B$) = A$ + " " + B$
150 CLS: PRINT TAB(15); "STRING DEFFN EXAMPLE"
160 PRINT: F$="": INPUT "ENTER FIRST NAME"; F$
165 IF F$="" THEN END
170 INPUT "ENTER LAST  NAME"; L$
180 '
190 ' ******* ADD F$ TO L$ WITH 1 BLANK IN BETWEEN *******
200 '
210 Z$ = FNADD$ (F$, L$)
220 PRINT TAB(6); "FULL NAME: "; Z$
230 GOTO 160
```

```
RUN ENTER
```

```
                STRING DEFFN EXAMPLE

ENTER FIRST NAME? JOHN ENTER
ENTER LAST  NAME? DOE ENTER
      FULL NAME: JOHN DOE
```

```
100 '          PROGRAM: MINMAX
110 ' EXAMPLE OF DEFFN FEATURE
120 '
130 ' ******* DEFINE MIN AND MAX FUNCTIONS *******
135 '
140 DEF FNMIN (A, B) = (A + B - ABS (A - B)) / 2
150 DEF FNMAX (A, B) = (A + B + ABS (A - B)) / 2
160 '
170 ' ******* READ 1ST VALUE - CALL IT THE MIN AND MAX *****
180 '          MN IS CURRENT MINIMUM VALUE
190 '          MX IS CURRENT MAXIMUM VALUE
200 '
210 READ MN: MX = MN
220 '
230 ' ******* GET NEXT VALUE AND FIND NEW MIN/MAX ********
240 '
250 READ V: IF V = 99999 THEN 320   'V=99999 MEANS ALL DONE
260 MN = FNMIN (MN, V)   'GET NEW MINIMUM
270 MX = FNMAX (MX, V)   'GET NEW MAXIMUM
280 GOTO 250
290 '
300 ' ******** PRINT RESULTS *******
310 '
320 PRINT "MINIMUM VALUE =", MN
330 PRINT "MAXIMUM VALUE =", MX
340 '
350 ' ******* DATA FOLLOWS - LAST VALUE MUST BE 99999 *******
360 '
370 DATA 1.2, 2, 3, 4.7, 5.332, 0.314, 6, 7, 8.3, 9.57, 99999
```

```
>RUN ENTER
MINIMUM VALUE =  .314
MAXIMUM VALUE =  9.57
READY
>370 DATA -1, 9, 0, 8, 1, 7, 2, 6, 3, 5, 4, 99999 ENTER
>RUN ENTER
MINIMUM VALUE = -1
MAXIMUM VALUE =  9
READY
>_
```

DEFUSR
(define entry address for USR routine)

DEFUSR*n*=*nmexp*

where *n* equals one of the digits 0,1,...,9; if *n* is omitted, 0 is assumed

nmexp specifies the entry address to a machine-language routine.

This statement lets you define the entry points for up to 10 machine-language routines. (In LEVEL II, where only one USR routine is available, the entry point address is POKEd into RAM.)

Example:

```
100 DEFUSR3=&H7D00
```

Assigns the entry point 7D00 hex, 32000 decimal, to the USR3 call. When your program calls USR3, control will branch to your sub-routine beginning at hex 7D00.

Here are three ways to get a machine language program into RAM so that it can be accessed via a USR*n* call:

1) Use the TRS-80 Editor Assembler, Radio Shack Catalog Number 26-2002, to convert the source code into an object file on tape; then load the tape under the SYSTEM command (use MEMORY SIZE to protect the code from destruction by BASIC).
2) Use the TRSDOS DEBUG program to type in the machine-code routine (then DUMP it to disk for safe-keeping); call DISK BASIC and answer MEMORY SIZE so as to protect the routine.
3) Have your DISK BASIC routine POKE the routine (decimal values for each byte) into high RAM. MEMORY SIZE should be set during initialization to protect the area you will POKE into.

See USR*n*.

INSTR (string search function)

INSTR([*n*,]*exp1$,exp2$*)

where *n* specifies a position in exp1$ where the search is to begin; if *n* is not supplied, 1 is assumed. (Position 1 is defined as the first character in the string.)
exp1$ is the string to be searched
exp2$ is the substring you want to search for

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise zero is returned. Note that the entire substring must be contained in the search string, or zero is returned. Also note that INSTR only finds the first occurrence of a substring, starting at the position you specify.

Examples (let A$="ABCDEFG"):

Expression	Result
INSTR(A$,"BCD")	2
INSTR(A$,"12")	0
INSTR(A$,"ABCDEFGH")	0
INSTR(3,"1232123","12")	5

See the EDIT program under MID$= for a sample use of INSTR.

LINE INPUT (input a line from keyboard)

LINE INPUT[*"prompt"*] ;*var$*

where *prompt* is a prompting message

var$ is the name that will be assigned to the line you type in

LINE INPUT (or LINEINPUT – the space is optional) is similar to INPUT, except:

- When the statement is executed, and the Computer is waiting for keyboard input, no question mark is displayed
- Each LINE INPUT statement can assign a value to just one variable
- Commas and quotes will be accepted as part of the string input
- Leading blanks are not ignored – they become part of *var$*
- The only way to terminate the string input is to press **ENTER**

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters – because only the **ENTER** key serves as a delimiter. If you want anyone to be able to input information to a program without special instructions, use LINE INPUT and then analyze the resultant string.

Some situations require that you input commas, quotes and leading blanks as part of the data. LINE INPUT serves well in such cases.

Examples:

```
LINE INPUT A$
```

Input A$ without displaying any prompt.

```
LINE INPUT"LAST NAME, FIRST NAME?";N$
```

Displays a prompt message and inputs data. Commas will not terminate the input string.

Try the following program to get the idea of LINE INPUT.

```
100 '         PROGRAM: LNINPUT
110 ' EXAMPLE OF LINEINPUT STATEMENT
120 '
130 CLEAR 300: CLS
140 PRINT TAB(15); "LINE INPUT STATEMENT": PRINT
150 PRINT: PRINT "*** ENTER TEXT ***"
151 '
152 ' *** GET STRING, THEN PRINT IT ***
153 '
155 A$=""   'SET A$ TO NULL STRING
```

```
160 LINEINPUT "==> "; A$
165 IF A$="" THEN END   'IF STILL NULL STRING, STOP!
170 PRINT A$
180 GOTO 155
```

```
RUN ENTER
```

```
                    LINE INPUT STATEMENT

*** ENTER TEXT ***
==>                 EXAMPLE TEXT
THIS TEXT HAS EMBEDDED LINE FEEDS AND TABS
IN      IT.  LINEINPUT ALSO ALLOWS DELIMITER (, :;"' ETC) ENTER
            EXAMPLE TEXT
THIS TEXT HAS EMBEDDED LINE FEEDS AND TABS
IN      IT.  LINEINPUT ALSO ALLOWS DELIMITER (, :;"' ETC).
==> ENTER
READY
>_
```

MID$= (replace portion of string)

MID$(*var$*,*n1*[,*n2*]) = *exp$*

where *var$* names the string to be changed
n1 specifies the starting position for the replacement
n2 specifies how many characters are to be replaced; if n2 is omitted, LEN(*exp$*) or LEN(*var$*)–*n1*+1 is used, whichever is smaller.

This statement lets you replace any part of a string with a specified substring, giving you a powerful string-editing capability.

Note that the length of the target string (*var$*) is never changed by the MID$= statement. If the replacement string, *exp$*, is too long to fit in the specified portion of var$, then the extra characters at the right of *exp$* are ignored.

However, if you specify the number of characters to be replaced, and this number is larger than the replacement string, then the length of the replacement string overrides the length you specified.

A$="ABCDEFG" at the beginning of each example below:

Ex. #	Expression	Resultant A$
1	MID$(A$,3,4)="12345"	AB1234G
2	MID$(A$,1,2)=""	ABCDEFG
3	MID$(A$,5)="12345"	ABCD123
4	MID$(A$,5)="01"	ABCD01G
5	MID$(A$,1,3)="***"	***DEFG

In example 2, the specified replacement length exceeds the length of the replacement string (which is zero); therefore the replacement-string length is used. In effect, no characters are replaced.

Sample program: EDIT

This program accepts an initial string, asks for a replacement position and a replacement string. Then it performs the MID$= replacement and prints the new string. Type in a position equal to zero to stop the program.

```
100 '          PROGRAM: EDIT
110 ' EXAMPLE OF INSTR FUNCTION FOR TEXT EDITTING
115 '
120 CLEAR 800: CLS
130 PRINT TAB(15); "  STRING-FUNCTION EDITOR"
135 '
140 ' ******* GET INITIAL TEXT *******
145 '
150 PRINT: PRINT "ENTER INITIAL TEXT STRING"
160 S$="": LINE INPUT S$: IF S$="" THEN END
165 '
170 ' ******* GET TARGET & REPLACEMENT STRINGS *******
175 '
180 T$="": PRINT: LINE INPUT"     TARGET STRING  "; T$
185 IF T$="" THEN END
190 LINE INPUT "REPLACEMENT STRING  "; R$
195 IF LEN(T$)<>LEN(R$)THEN PRINT"CAN'T CHANGE STRING LENGTH":
    GOTO 180
200 ' ******* MAKE REPLACEMENT(S) AND PRINT NEW STRING *****
210 I=1   'VARIABLE I POSITIONS TO BEGINNING POINT OF SEARCH
220 I=INSTR(I, S$, T$): IF I=0 THEN 150   'I=0 IF NOT FOUND
230 MID$(S$,I)=R$   'MAKE REPLACEMENT
240 PRINT "POSITION - "; I: PRINT S$
250 I=I+LEN(R$): GOTO 220   'ADVANCE POSITION
```

```
RUN ENTER

                STRING-FUNCTION EDITOR

ENTER INITIAL TEXT STRING
CHANGE "DISC" TO "DISK" EACH TIME IT OCCURS. . .(DISC=>DISK)

     TARGET STRING  DISC ENTER
REPLACEMENT STRING  DISK ENTER
POSITION -  9
CHANGE "DISK" TO "DISK" EACH TIME IT OCCURS. . .(DISC=>DISK)
POSITION -  48
CHANGE "DISK" TO "DISK" EACH TIME IT OCCURS. . .(DISK=>DISK)

ENTER INITIAL TEXT STRING ENTER

READY
>_
```

TIME$ (get value of Real-Time Clock)

TIME$

TIME$ is a function with no arguments – when executed, it returns a string-value composed of the date and time currently stored in the Real-Time Clock memory area. The string is always 17 characters long and has the following format:

MM/DD/YY␣HH:MM:SS (month/day/year hr:min:sec)

The hour appears in 24-hour form, e.g., 1:30 PM appears as 13:30.

To set the time and date, get into the DOS READY mode and use the TRSDOS commands, TIME and DATE, as follows (assume it's 3:30 PM on January 1, 1979):

```
TIME 15:30:00 ENTER
DATE 01/01/79 ENTER
```

Or, you can set the time and date under DISK BASIC, by POKEing the time and date values into the appropriate addresses (see CLOCK, **TRSDOS Library Commands**).

TIME$ can be printed or used internally by your program in dedicated applications.

Examples:

```
1000 IF LEFT$(TIME$,15)="07/04/79 20:00"THEN 2000
1010 GOTO 1000
2000 REM...IT'S 8PM ON JULY 4TH, 1979
2010 REM...START FIREWORKS DISPLAY
.
.
.
```

The following program, CLOCK, will display the time and date until you press the @-key.

```
100 '          PROGRAM: CLOCK
110 ' EXAMPLE OF TIME$
120 '
130 CLS: PRINT CHR$ (23)   'GET INTO 32 CHARACTER MODE
140 '
150 ' ******* PRINT TIME AND DATE *******
160 '
170 PRINT @ 264, "THE TRS-80 TIME IS";
180 PRINT @ 458, "DATE: ";  LEFT$ (TIME$, 8);
190 PRINT @ 586, "TIME: "; RIGHT$ (TIME$, 8);
200 '
210 ' ******* STOP IF "@" KEY IS DEPRESSED *******
220 '
230 A$=INKEY$: IF A$ = "@" THEN END ELSE 180
```

USRn (call to user's external subroutine)

USR[*n*](*nmexp*)

where *n* specifies one of ten available USR calls, *n*=0,1,2,...,9. If *n* is omitted, zero is assumed.

nmexp is in the range < –32768 +32767> and is passed as an integer argument to the routine

These functions (USR0 through USR9) transfer control to machine-language routines previously defined with DEFUSR*n* statements.

When a USR call is encountered in a statement, control goes to the address specified in the DEFUSR*n* statement. This address specifies the entry point to your machine-language routine. A RET or JP 0A9A instruction in the routine returns control to the USR call in your BASIC program.

Note: If you call a USR*n* routine before defining the routine entry point with DEFUSR*n*, an ILLEGAL FUNCTION CALL error will occur.

You can pass one argument and retrieve one output value directly via the USR argument; or you can pass and retrieve arguments indirectly via POKE and PEEK statements.

Example:

```
10 DEFUSR1=&H7D00
20 REM...MORE PROGRAM LINES HERE
100 A=USR1(X)
```

The effect of this sequence is to:

1) Define USR1 as a routine with an entry point at hex 7D00 (line 10)
2) Transfer control to the routine; the value X can be passed to the routine if the routine makes the CALL described below (line 100)
3) When the routine returns to BASIC, the variable A may contain the value passed back from the routine (if your routine makes the JUMP described below); otherwise A will be assigned the value of X (line 100).

Passing arguments to and from USR routines

There are several ways to pass arguments back and forth between your BASIC main program and your USR routines: the two major ways are listed below.

1. POKE the argument(s) into fixed RAM locations. The machine-language routine can then access these values and place results in other RAM locations. When the routine returns control to BASIC, your program can PEEK into these addresses to pick up the "output" values. **This is the only way to pass two or more arguments back and forth.**

2. Pass one argument to the routine as the argument in the USRn call, then use special ROM calls to access this argument and return a value to BASIC. **This method is limited to sending one argument and returning one value** (both are integers).

ROM Calls

CALL OA7FH — Puts the USR argument into the HL register pair; H contains msb, L contains lsb. This CALL should be the first instruction in your USR routine.

JP 0A9AH — Use this JUMP to return to BASIC; the integer in HL becomes the output of the USR call. If you don't care about returning HL, then execute a simple RETurn instruction instead of this JUMP.

Examples:

Listed below is an assembled machine-language routine that will accept the argument from the USR call in BASIC, left-shift it one position, and return the result to BASIC.

```
                00100 ;
                00110 ; SHIFT FUNCTION
                00120 ;
                00130 ;       MACHINE CODE PROGRAM TO LEFT SHIFT
                00140 ;       AN ARGUMENT SENT FROM BASIC AND RETURN
                00150 ;       THE RESULT BACK TO BASIC
                00160 ;
7D00            00170         ORG     7D00H
                00180 ;
                00190 ; EQUATES AND ENTRY POINTS
                00200 ;
0A7F            00210 GETARG  EQU     0A7FH          ;GET ARGUMENT FROM BASIC
0A9A            00220 PUTANS  EQU     0A9AH          ;RETURN ANSWER TO BASIC
                00230 ;
7D00 CD7F0A     00240 SHIFT   CALL    GETARG         ;GET NUMBER FROM BASIC
7D03 CB15       00250         RL      L              ;SHIFT L
7D05 CB14       00260         RL      H              ;SHIFT H - ANSWER IN HL
7D07 C39A0A     00270         JP      PUTANS         ;RETURN TO BASIC W/ ANSWER
                00280 ;
7D00            00290         END     SHIFT
```

The following program includes the decimal code for the SHIFT routine. The code is POKEd into RAM and then accessed as a USR routine. RUN the program; to stop, enter a value of zero.

Note: The following two BASIC programs require that you reserve memory addresses above 31999 for the USR Code. (Answer MEMORY SIZE? with 31999.)

```
100 '         PROGRAM: SHIFT
110 ' MACHINE LANGUAGE USER FUCTION TO LEFT SHIFT
120 '
130 ' ******* MACHINE CODE AT 7D00 HEX *******
140 '
150 DEFUSR5 = &H7D00
160 '
170 ' ******* POKE USER PROGRAM INTO MEMORY *******
180 '
190 FOR X = 32000 TO 32009   '7D00 HEX EQUALS 32000 DECIMAL
200     READ A
210     POKE X,A
220 NEXT X
230 '
240 ' ******* GET VALUE FROM USER *******
250 '
260 CLS: PRINT TAB(15); "USR5 LEFT-SHIFT FUNCTION"
270 PRINT: INPUT"ENTER INTEGER VALUE"; V
280 IF V=0 THEN END
290 PRINT "LEFT SHIFTED VALUE = "; TAB(32); USR5(V)
300 GOTO 270
310 '
320 ' ******* DATA IS DEMICAL CODE FOR HEX PROGRAM *******
330 '
340 DATA 205,127,10,203,21,203,20,195,154,10
```

```
RUN ENTER
```

```
                USR5 LEFT-SHIFT FUNCTION

ENTER INTEGER VALUE? 7 ENTER
LEFT SHIFTED VALUE =              14

ENTER INTEGER VALUE? -7 ENTER
LEFT SHIFTED VALUE =             -13

ENTER INTEGER VALUE? 32767 ENTER
LEFT SHIFTED VALUE =              -2

ENTER INTEGER VALUE? 0 ENTER
READY
>_
```

Listed below is an assembled program to white out the display (an "inverse" CLEAR key!).

```
                00100 ;
                00110 ; ZAP OUT SCREEN USR FUNCTION
                00120 ;
7D00            00130         ORG     7D00H
                00140 ;
                00150 ; EQUATES
                00160 ;
3C00            00170 VIDEO   EQU     3C00H           ;START OF VIDEO RAM
00BF            00180 WHITE   EQU     0BFH            ;ALL WHITE GRAPHICS BYTE
03FF            00190 COUNT   EQU     3FFH            ;NUMBER OF BYTES TO MOVE
                00200 ;
                00210 ; PROGRAM CHAIN MOVES X'BF' INTO ALL OF VIDEO RAM
                00220 ;
7D00 21003C     00230 ZAP     LD      HL,VIDEO        ;SOURE ADDRESS
7D03 36BF       00240         LD      (HL),WHITE      ;PUT OUT 1ST BYTE
7D05 11013C     00250         LD      DE,VIDEO+1      ;DESTINATION ADDRESS
7D08 01FF03     00260         LD      BC,COUNT        ;NUMBER OF ITERATIONS
7D0B EDB0       00270         LDIR                    ;DO IT TO IT!!!
                00280 ;
7D0D C9         00290         RET                     ;RETURN TO BASIC
7D00            00300         END     ZAP
```

This routine can be POKEd into RAM and accessed as a USR routine, as follows.

```
100 '          PROGRAM: USR1
110 ' EXAMPLE OF A USER MACHINE LANGUAGE FUNCTION
115 ' DEPRESS THE '@' KEY WHILE NUMBERS ARE PRINTING TO STOP
120 '
130 ' ******* POKE MACHINE PROGRAM INTO MEMORY *******
140 '
150 DEFUSR1 = &H7D00
160 FOR X = 32000 TO 32013   '7D00 HEX EQUAL 32000 DECIMAL
170     READ A
180     POKE X, A
190 NEXT X
192 '
194 ' ******* CLEAR SCREEN & PRINT NUMBERS 1 THRU 100 *******
196 '
200 CLS
205 PRINT TAB(15); "WHITE-OUT USER ROUTINE": PRINT
210 FOR X = 1 TO 100
220     PRINT X;
225     A$ = INKEY$: IF A$ = "@" THEN END
230 NEXT X
240 '
250 ' ******* JUMP TO WHITE-OUT SUBROUTINE *******
260 '
270 X = USR1 (0)
280 FOR X = 1 TO 1000: NEXT X   'DELAY LOOP
290 GOTO 200
300 '
310 ' ******* DATA IS DEMICAL CODE FOR HEX PROGRAM *******
320 '
330 DATA 33,0,60,54,255,17,1,60,1,255,3,237,176,201
```

RUN the program. An equivalent BASIC white out routine takes a long time by comparison!

Disk-Related Features

Programs and data are stored as "files" under TRSDOS. Each program or data-set on the disk has its own, distinct file specification – which includes a name plus identifying information.

Before attempting any disk input/output – including loading and saving BASIC programs, refer to the **TRSDOS Overview**. Also, review the Notation Conventions described under **General Information**. That's the only way to be sure you understand the statement syntax descriptions.

DISK BASIC provides a powerful set of commands, statements and functions relating to disk I/O under TRSDOS. These fall into two categories:

1. File manipulation: dealing with files as units, rather than with the distinct records the files contain.

2. File access: preparing data files for I/O; reading and writing to the files.

Commands discussed under "File Manipulation":

KILL	delete a program or data file from the disk
LOAD	load a BASIC program from disk
MERGE	merge an ASCII-format BASIC program on disk with one currently in RAM
RUN"*program*"	load and execute a BASIC program stored on disk
SAVE	save the resident BASIC program on disk

Statement and functions discussed under "File Access":

Statements

OPEN	Open a file for access (create the file if necessary)
CLOSE	Close access to the file
INPUT #	Read from disk, sequential mode
LINE INPUT#	Read a line of data, sequential mode
PRINT#	Write to disk, sequential mode
GET	Read from disk, random access mode
PUT	Write to disk, random access mode
FIELD	Assign field sizes and names to random access file buffer
LSET	Place value in specified buffer field, add blanks on the right to fill field
RSET	Place value in specified buffer field, add blanks on the left to fill field

Functions

CVD	Restore double-precision number to numeric form after GETting from disk
CVI	Restore integer to numeric form after GETting from disk
CVS	Restore single-precision number to numeric form after GETting from disk
EOF	Check to see if end of file encountered during read
LOF	Return number of last record in file
MKD$	Convert double-precision number to string so it can be PUT on disk
MKI$	Convert integer to string so it can be PUT on disk
MKS$	Convert single-precision number to string so it can be PUT on disk

File Manipulation

KILL (delete a file from the disk)

KILL *exp$*

where *epx$* defines a file specification for an existing file

This command works like the TRSDOS KILL command – see **TRSDOS Library Commands.**

Example:

```
KILL"OLDFILE/BAS. PSW1
```

deletes the file specified from the first drive which contains it.

Do not KILL an open file, or you may destroy the contents of the diskette. (First CLOSE the open file.)

LOAD (load BASIC program file from disk)

LOAD *exp$* [,R]

where *exp$* defines a filespec for a BASIC program file stored on disk

R tells BASIC to RUN the program after it is loaded

This command loads a BASIC program file into RAM; if the R option is used, BASIC will proceed to RUN the program automatically; otherwise, BASIC will return to the command mode.

LOAD without the R option wipes out any resident BASIC program, clears all variables, and closes all open files. LOAD with the R option deletes the resident program and clears all variables, but does not close the open files.

LOAD with the R option is equivalent to the command RUN *exp$*,R. Either of these commands can be used inside programs to allow program chaining – one program calling another, etc.

If you attempt to LOAD a non-BASIC file, a DIRECT STATEMENT IN FILE or LOAD FORMAT ERROR will occur.

Examples:

LOAD"PROG1/BAS:2"	Clears resident BASIC program and loads PROG1/BAS from drive 2; returns to BASIC command mode.
10 REM . . . INSTRUCTIONS	Example of chaining two programs – the first may be used to give instructions and then to load the "working" part of the program (PROG2/BAS). Note that line 1000 is equivalent to: 1000 RUN"PROG2/BAS"
1000 LOAD"PROG2/BAS",R	

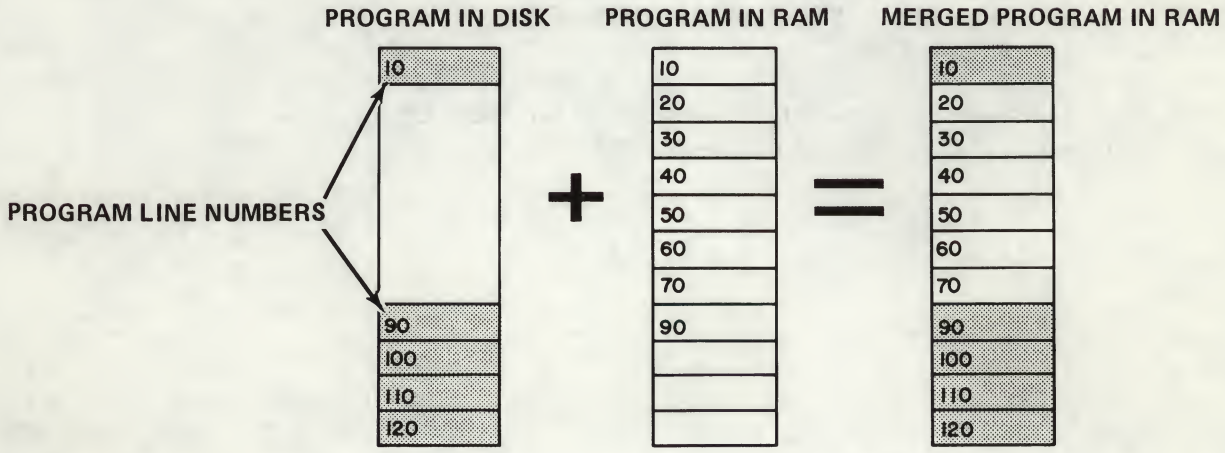
MERGE
(merge disk program with resident program)

MERGE *exp$*

where *exp$* defines a filespec for an ASCII-format BASIC disk file, e.g., a program saved with the A-option.

MERGE is similar to LOAD – except that the resident program is not wiped out before the new program *exp$* is loaded. Instead, *exp$* is merged into the resident program.

That is, program lines in *exp$* will simply be inserted into the resident program in sequential order. If line numbers in *exp$* coincide with line numbers in the resident program, the resident lines will be replaced by those from *exp$*.



MERGE provides a convenient means of putting modular programs together. For example, an often-used set of BASIC subroutines can be tacked onto a variety of programs with this command.

For example, suppose the following program is in RAM:

```
10 REM...MAIN PROGRAM
20 GOSUB 1000
30 REM...MORE PROGRAM LINES HERE
999 END
1000 REM...NEED TO ADD SUBROUTINES HERE
1010 REM...SO USE MERGE COMMAND
1020 PRINT"SUBROUTINE NOT AVAILABLE":RETURN
```

And suppose the following program is stored on disk in ASCII format:

```
1000 REM...BEGINNING OF SUBROUTINE
1010 PRINT"EXECUTING SUBROUTINE..."
1020 REM...MORE PROGRAM LINES HERE
1100 RETURN
```

Assuming the subroutine program is named SUB/TXT, then we could MERGE it with the statement:

```
MERGE"SUB/TXT"
```

and the resultant program in RAM would be:

```
10 REM...MAIN PROGRAM
20 GOSUB 1000
30 REM...MORE PROGRAM LINES HERE
999 END
1000 REM...BEGINNING OF SUBROUTINE
1010 PRINT"EXECUTING SUBROUTINE..."
1020 REM...MORE PROGRAM LINES HERE
1100 RETURN
```

Note that MERGE closes all files and clears all variables. Upon completion, BASIC returns to the command mode.

RUN"program"
(load and execute a program from disk)

RUN *exp$* [,R]

where *exp$* defines the filespec for a BASIC program stored on disk. R leaves open files open

If the R-option is not selected, all open files will be closed.

When the command is executed, any resident BASIC program will be replaced by the program contained in *exp$*.

Example:

```
RUN"DISKDUMP/BAS" ENTER
```

Loads and executes the BASIC sector-dump program.

Suppose you save the following program on disk with the name "PROG1/BAS":

```
10 PRINT"PROG1 EXECUTING..."
20 RUN"PROG2/BAS"
```

And save this program on disk with the name "PROG2/BAS":

```
10 PRINT"PROG2 EXECUTING..."
20 RUN"PROG1/BAS"
```

Now type:
`RUN"PROG1/BAS ENTER`
and you'll see a simple example of program chaining.
Hold down the BREAK key to interrupt the program chain.

SAVE (save program onto disk)

SAVE *exp$* [,A]

where *exp$* defines the file-name and optional extension, password, and drive to be used. If the file-name already exists, its previous contents will be lost as the file is re-created.

A causes the file to be stored in ASCII rather than compressed-format.

This command lets you save your BASIC programs on disk. You can save the program in compressed or ASCII format.

Using compressed-format takes up less disk space and is faster during both SAVEs and LOADs. This is the way BASIC programs are stored in RAM.

Using the ASCII option makes it possible to do certain things that cannot be done with compressed-format BASIC files.

Examples:

- The MERGE command requires that the disk file be in ASCII form.
- You can use the TRSDOS commands LIST and PRINT with ASCII-format files.
- Programs which read in other programs as data will typically require that the data programs be stored in ASCII.

Useful conventions for placing extensions on BASIC programs:
For compressed-format programs, use the extension /BAS.
For ASCII format programs, use the extension /TXT.

Examples of SAVE command:

```
SAVE"FILE1/BAS.JOHNQDOE:3"
```
saves the resident BASIC program in compressed-format with the file name FILE1, extension /BAS, password .JOHNQDOE; the file is placed on drive :3.

```
SAVE"MATHPAK/TXT",A
```
saves the resident program in ASCII form, using the name MATHPAK/TXT, on the first non write-protected drive.

Upon completion of a SAVE, BASIC returns in the command mode.

File Access

This section is divided into four parts:

1) Creating files and assigning buffers – OPEN and CLOSE
2) Statements and functions
3) Sequential I/O techniques
4) Random I/O techniques.

If this is your first experience with disk file access, you should concentrate on parts 1, 3 and 4, perhaps just skimming through part 2 to get a general idea of how the functions and statements work. Later you can go back to part 2 and learn the details of statement and function syntax.

Creating files and assigning buffers

During the initialization dialog, you type in a number in response to HOW MANY FILES? The number you type in tells BASIC how many buffers to create to handle your disk accesses (reads and writes).

Each buffer is given a number from 1 to 15. If you type:

```
HOW MANY FILES? 4 ENTER
```

then BASIC sets aside four buffers, numbered 1,2,3 and 4.

You can think of a buffer as a waiting area that data must pass through on the way to and from the disk file. When you want to access a particular file, you must tell BASIC which buffer to use in accessing that file. You must also tell BASIC what kind of access you want – sequential output, sequential input, or random input/output.

All this is done with the OPEN statement, and "undone" with the CLOSE statement.

OPEN
(Assign a buffer to a file and set mode)

OPEN *exp1$,nmexp,exp2$*

where *exp1$* is a string expression or constant of which only the first character is significant; this character specifies the mode in which the file is to be opened:

exp1$=	access mode
I	sequential input
O	sequential output
R	random I/O

nmexp has a value from 1 to 15, and tells BASIC which buffer to assign to the file specified by *exp2$*

exp2$ defines a TRSDOS file specification

This statement makes it possible to access a file. *exp1$* determines what kind of access you'll have via the specified buffer; *nmexp* determines which buffer will be assigned to the file; and *exp2$* names the file to be accessed. If *exp2$* does not exist, then TRSDOS may or may not create it, depending on the access mode.

Note: *nmexp* (buffer number) cannot exceed the number you entered for the FILES? question during initialization. If you entered:

```
HOW MANY FILES? 2 ENTER
```

then *nmexp* can have the value 1 or 2.

Examples of OPEN statements:

```
100 OPEN "O",1,"CLIENTLS/TXT"
```

Opens the file "CLIENTLS/TXT" for sequential output. Buffer 1 will be used. If the file does not exist, it will be created. If it already exists, then its previous contents are lost. (This is explained under "Sequential I/O Techniques".)

```
100 OPEN "I",1,"PROG1/TXT:1"
```

Opens the file "PROG1/TXT" on drive 1 for sequential input. Buffer 2 is assigned to the file. If PROG1/TXT does not exist on drive 1, an error message is returned – since you can't input from a non-existent file!

```
100 INPUT"MODE (I,O,R)";MODE$
110 INPUT"BUFFER NUMBER";BUFFER%
120 INPUT"FILE SPECIFICATION";FILESPEC$
130 OPEN MODE$,BUFFER%,FILESPEC$
```

This sequence of statements lets you provide the arguments for the OPEN statement during program execution. The first character of MODE$ sets the access mode, BUFFER% determines which buffer will be used, and FILESPEC$ gives the file specification.

```
OPEN"R",2,"DATA/BAS.SPECIAL"
```

Opens the file DATA/BAS with password SPECIAL, in the random I/O mode, using buffer number 2. If DATA/BAS does not exist, it will be created on the first non write-protected drive.

While a file is open, it is referenced by the buffer-number which was assigned to it. Examples:

GET *buffer-number*
PUT *buffer-number*
PRINT#*buffer-number*
INPUT#*buffer-number*

All these statements will reference the file which was OPENed via *buffer-number*. **The mode must be correct.**

Once a buffer has been assigned to a file with the OPEN statement, that buffer cannot be used in another OPEN statement. You have to CLOSE it first.

More on Buffer Assignments

Two or more buffers may be assigned to the same file for sequential input (I-mode). However, only one buffer at a time may be assigned to a file for sequential output (O-mode) or random access R-mode.

For example:

```
10 OPEN "I",1,"TEST/TXT:1"
20 OPEN "I",2,"TEST/TXT:1"
```

Now TEST/TXT can be accessed via buffers 1 and 2 for sequential input.

CLOSE (close access to the file)

CLOSE [*nmexp* [,*nmexp* . . .]]

where *nmexp* has a value from 1 to 15, and refers to the file's buffer-number (assigned when the file was opened). If *nmexp* is omitted, all open files will be closed.

This command terminates access to a file through the specified buffer(s). If *nmexp* has not been assigned in a previous OPEN statement, then

CLOSE *nmexp*

has no effect.

Examples of CLOSE statements:

```
CLOSE 1,2,8
```

Terminates the file assignments to buffers 1, 2 and 8. These buffers can now be assigned to other files with OPEN statements.

```
CLOSE FIRST%+COUNT%
```

Terminates the file assignment to the buffer specified by the sum (FIRST% + COUNT%).

Do not remove a diskette which contains an open file – first close the file. This is because the last 256 bytes of data may not have been written to disk yet. Closing the file will write the data, if it hasn't already been written.

The following actions and conditions cause all files to be automatically closed:

NEW ENTER
RUN ENTER
MERGE *filespec* ENTER
EDITing a file
Adding or deleting program lines
Execution of the CLEAR *n* statement
Disk Errors

INPUT# (sequential read from disk)

INPUT# *nmexp*, *var*[,*var*...]

where *nmexp* specifies a sequential input file buffer, *nmexp*=1,2,...,15

var is the variable name to contain the data from the file

This statement inputs data from a disk file. The data is input sequentially. That is, when the file is first opened, a pointer is set to the beginning of the file. Each time data is input, the pointer advances. To start over reading from the beginning of the file, you must close the file-buffer and re-open it.

INPUT# doesn't care how the data was placed on the disk – whether a single PRINT# statement put it there, or whether it required 10 different PRINT# statements. What matters to INPUT# are the positions of the terminating characters and the EOF marker.

To INPUT# data successfully from disk, you need to know ahead of time what the format of the data is. Here is a description of how INPUT# interprets the various characters it encounters when reading data.

When inputting data into a variable, BASIC ignores leading blanks; when the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The particular terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

Special Note

Here's an important exception to keep in mind in reading the following material.

When <EN> (a carriage return) is preceded by <LF> (a line feed), the <EN> is not taken as a terminator. Instead, it becomes a part of the data item (string variable) or is simply ignored (numeric variable).

(To enter the <LF> character from the keyboard, press the down arrow. To enter the <EN> character, press ENTER.)

This exception applies to all cases noted below where <EN> is said to be a terminator.

Numeric Input

Suppose the data image on disk is

␣1.234␣-33␣␣27␣ <EN>

<EN> denotes a carriage-return character (ASCII code decimal 13).

Then the statement

```
INPUT#1, A,B,C
```

or the sequence of statements

```
INPUT#1,A: INPUT#1,B: INPUT#1,C
```

will assign the values as follows:

A=1.2345
B=–33
C=27

This works because blanks and <EN> serve as terminators for input to numeric variables. The blank before 1.2345 is a "leading blank", therefore it is ignored. The blank after 1.2345 is a terminator; therefore BASIC starts inputting the second variable at the – character, inputs the number –33, and takes the next two blanks as terminators. The third input begins at the 2 and ends with the 7.

String Input

When reading data into a string variable, INPUT ignores all leading blanks; the first non-blank character is taken as the beginning of the data item.

If this first character is a double-quote ("), then INPUT will evaluate the data as a quoted string: it will read in all subsequent characters up to the next double-quote. Commas, blanks, and < EN > –characters will be included in the string. The quotes themselves do not become a part of the string.

If the first character of the string item is not a double-quote, then INPUT will evaluate the data as an unquoted string: It will read in all subsequent characters up to the first comma, or < EN > .
If double quotes are encountered, they will be included in the string.

For example, if the data on disk is:

PECOS,␣TEXAS "GOOD MELONS"

Then the statement

```
INPUT#1, A$, B$, C$
```

would assign values as follows:

A$=PECOS
B$=␣TEXAS "GOOD␣MELONS"
C$= null string

If a comma is inserted in the data image before the first double quote, C$ will get the value, GOOD MELONS.

These are very simple examples just to give you an idea of how INPUT works. However, there are many other ways to input data – different terminators, different target variable types, etc.

Rather than taking a shotgun approach and trying to cover them all, we'll give a generalized description of how input works and what the terminating characters and conditions are, and then provide several examples.

When BASIC encounters a terminating character, it scans ahead to see how many more terminating characters it can include with the first terminator. This ensures that BASIC will begin looking for the next data item at the correct place.

The list below defines the various terminating sets INPUT# will look for. It will always try to take-in **the largest set possible.**

Numeric-input terminator sets

end of file encountered
255th data character encountered
, (comma)
<EN>
<EN> <LF>
␣[␣...][<EN>]
␣[␣...][<EN><LF>]

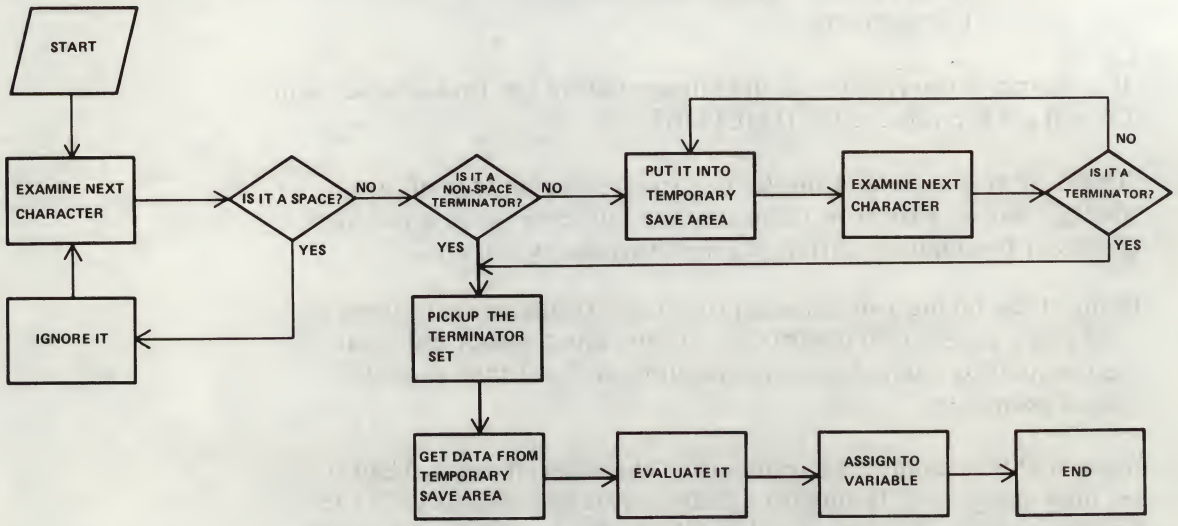
Quoted-string terminator sets

end of file encountered
255th data character encountered
" (double quote)
" [␣...][,]
" [␣...][<EN>]
" [␣...][<EN> <LF>]

Unquoted-string terminator sets

end of file encountered
255th data character encountered
,
<EN> [<LF>]

Here's a flow chart describing how INPUT# assigns data to a variable:



The following table shows how various data images will be read-in by the statement:

```
INPUT#1, A, B, C
```

Ex.#	Image on disk	Values assigned
1	␣123.45␣ <EN><LF>␣8.2E4␣␣7000<EN>	A=123.45 B=82000 C=7000
2	␣␣␣3<LF><EN> 4 <EN>5 <EN> A12eof	A=34 B=5 C=0
3	1,,2,3,4 <EN>	A=1 B=0 C=2
4	1,3,end-of-file	A=1 B=2 C=0 end of file error

In Example 2 above, why does variable C get the value 0? When the input reaches the end of file, it terminates the last data item, which then contains "A12". This is evaluated by a routine just like the BASIC VAL function –which returns a zero since the first character of "A12" is non-numeric.

In Example 3, when INPUT# goes looking for the second data item, it immediately encounters a terminator (the comma); therefore variable B is given the value zero.

The following table shows how various data images on disk will be read by the statement:

```
INPUT#1, A$, B$
```

Ex.#	Image on disk	Values assigned
1	␣␣␣"ROBERTS,J."ROBERTS,M.N eof	A$=ROBERTS,J. B$=ROBERTS,M.N.
2	␣␣␣ROBERTS,J.,␣␣␣ROBERTS,M.N. <EN>	A$=ROBERTS B$=J.
3	THE WORD "QUO",12345.789 <EN>	A$=THE WORD "QUO" B$=12345.789
4	BYTE<LF> <EN> UNIT OF MEMORY eof	A$=BYTE<LF> <EN> UNIT OF MEMORY B$=null (eof error)

In example 3, the first data item is an unquoted string, therefore the double-quotes are not terminators, and become part of A$.

In example 4, the <EN> is preceded by an <LF>, therefore it does not terminate the first string; both <LF> and <EN> are included in A$.

Technical Note: The above discussion ignores the role of the input buffer in the sequential input process. Actually, DISK BASIC always reads in 256-byte data records into the buffer, and then sorts through what's in the buffer to "satisfy" the INPUT# variable list. That's why

```
100 INPUT#1, A%
200 INPUT#1, B%
```

do not necessarily require two disk accesses. The 256-byte record in the buffer can contain enough data for A%, B% and more.

LINE INPUT# (read a line of text from disk)

LINE INPUT#*nmexp*,*var$*

where *nmexp* specifies a sequential output file buffer, *nmexp*=1,2,...,15

var$ is the variable name to contain the string data

Similar to LINE INPUT from keyboard, this statement reads a "line" of string data into var$. This is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT (or LINEINPUT – the space is optional) reads everything from the first character up to:

1) an <EN> character which is not preceded by <LF>
2) the end-of-file
3) the 255th data character (this 255 character is included in the string)

Other characters encountered – quotes, commas, leading blanks, <LF> <EN> pairs – are included in the string.

For example, if the data looks like:

```
10 CLEAR 500   <EN>
20 OPEN "I",1,"PROG"   <EN>
.
.
.
```

then the statement

```
LINEINPUT#1,A$
```

could be used repetitively to read each program line, one line at a time.

PRINT# (sequential write to disk file)

PRINT#*nmexp*,[USING *format$*;] *exp*[*p exp* . . .]

where *nmexp* specifies a sequential output file buffer, *nmexp*=1,2,. . .,15

format$ is a sequence of field specifiers used with the USING option

p is a delimiter placed between every two expressions to be PRINTed to disk; either a semi-colon or comma can be used (semi-colon is preferable)

exp is the expression to be evaluated and written to disk

This statement writes data sequentially to the specified file. When you first open a file for sequential output, a pointer is set to the beginning of the file, therefore your first PRINT# places data at the beginning of the file. At the end of each PRINT# operation, the pointer advances, so the values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to display creates on the screen. Remember this, and you'll be able to set up your PRINT# list correctly for access by one or more INPUT statements.

PRINT# does not compress the data before writing it to disk; it writes an ASCII-coded image of the data.

For example, if A=123.45

```
PRINT#1, A
```

will write a nine-byte character sequence onto disk:

␣123.45␣ <EN>

The punctuation in the PRINT list is very important. Unquoted commas and semi-colons have the same effect as they do in regular PRINT to display statements.

For example, if A=2300 and B=1.303, then

```
PRINT#1, A, B
```

places the data on disk as

␣2300␣␣␣␣␣␣␣␣␣␣1.303␣ <EN>

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semi-colons instead of commas.

```
PRINT#1, A; B
```

writes the data as:

2300 1.303 <EN>

PRINT# with numeric data is quite straightforward – just remember to separate the items with semi-colons.

PRINT# with string data requires more care, primarily because you have to insert delimiters so the data can be read back correctly. In particular, you must separate string items with explicit delimiters if you want to INPUT# them as distinct strings.

For example, suppose:

A$="JOHN Q. DOE" and B$="100-01-001"

Then:

```
PRINT#1, A$; B$
```

would produce this image on disk:

JOHN Q. DOE100-01-001 <EN>

which could not be INPUT back into two variables.

The statement:

```
PRINT#1, A$;",";B$
```

would produce:

JOHN Q. DOE, 100-01-001

which could be INPUT# back into two variables.

This method is adequate if the string data contains no delimiters – commas or <EN> –characters. But if the data does contain delimiters or leading blanks that you don't want to ignore, then you must supply explicit quotes to be written along with the data. For example, suppose A$="DOE, JOHN Q." B$="100-01-001"

If you use

```
PRINT#1,A$;",";B$
```

the disk image will be:

DOE, JOHN Q.,100-01-001 <EN>

When you try to input this with a statement like

```
INPUT#2,A$,B$
```

A$ will get the value "DOE", and B$ will get "JOHN Q." – because of the comma after DOE in the disk image.

To write this data so that it can be input correctly, you must use the CHR$ function to insert explicit double quotes into the disk image. Since 34 is the decimal ASCII code for double quotes, use CHR$(34) as follows:

```
PRINT#1,CHR$(34);A$;CHR$(34);B$
```

this produces the disk image

"DOE, JOHN Q."100-01-001 <EN>

which can be read with a simple

```
INPUT#2,A$,B$
```

Note: You can also use the CHR$ function to insert other delimiters and control codes into the file, for example:

CHR$(10)	<LF> Line Feed
CHR$(13)	carriage return (<EN>character)
CHR$(11) or CHR$(12)	line-printer top-of-form

USING Option

This option makes it easy to write files in a carefully controlled format. You could create a report file this way, which then could be LISTed or PRINTed (TRSDOS commands).

Or you could use this option to control how many characters of a value are written to disk.

For example, suppose:

```
A$="LUDWIG"
B$="VAN"
C$="BEETHOVEN"
```

Then the statement

```
PRINT#1,USING"!.!.%  %";A$;B$;C$
```

would write the data in nickname form:

```
L.V.BEET    <EN>
```

(In this case, we didn't want to add any explicit delimiters.) See the PRINT USING description in the LEVEL II BASIC Reference Manual for a complete explanation of the field-specifiers.

Technical Note: The above discussion ignores the role of the output buffer in the sequential write process. Actually, the data is first placed into the buffer, and then, as 256-byte records are filled, the data is written to the disk file. That's why there isn't always a disk access during execution of each PRINT# statement.

Random Access Statements

FIELD
(organize a random file-buffer into fields)

FIELD *nmexp,nmexp1* AS *var1$* [,*nmexp2* AS *var2$* . . .]

where *nmexp*	specifies a random access file buffer, *nmexp*=1,2,. . . ,15
nmexp1	specifies the length of the first field,
var1$	defines a variable name for the first field
nmexp2	specifies the length of the second field
var2$	defines a variable name for the second field
. . .	subsequent *nmexp* AS *var$* pairs define other fields in the buffer

Before FIELDing a buffer, you must use an OPEN statement to assign that buffer to a particular disk file (must use random access mode). Then use the FIELD statement to organize a random file buffer so that you can pass data from BASIC to disk storage and vice-versa.

Each random file buffer has 255 bytes which can store data for transfer from disk storage to BASIC or from BASIC to disk. However, you need a way to access this buffer from BASIC so that you can either read the data it contains or place new data in it. The FIELD statement provides the means of access.

You may use the FIELD statement any number of times to "re-organize" a file buffer. FIELDing a buffer does not clear the contents of the buffer; only the means of accessing the buffer (the field names) are changed. Furthermore, two or more field names can reference the same area of the buffer.

Examples:

```
FIELD 1, 255 AS A$
```

This statement tells BASIC to assign the entire 255-byte buffer to the string variable A$. If you now print A$, you will see the contents of the buffer. Of course, this value would be meaningless unless you have used GET to read a 255-byte record from disk.

Note: All data – both strings and numbers – must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI,MKS$/CVS,MKD$/CVD) for converting numbers to strings and vice-versa. See "Functions", below.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS ST$,7 AS ZP$
```

The first 16 bytes of buffer 3 are assigned the buffer name NM$; the next 25, AD$; the next 10, CY$; the next 2, ST$; and the next 7, ZP$. The remaining 195 bytes of the buffer are not fielded at all.

More on field names

Field names, like NM$,AD$,CY$,ST$ and ZP$, are not string variables in the ordinary sense. They do not consume the string space available to BASIC.

Instead, they point to the buffer field which you assigned with the FIELD statement. That's why you can use:

```
100 FIELD 1,255 AS A$
```

without worrying about whether 255 bytes of string space are available for A$.

If you use a buffer field name on the left side of an ordinary assignment statement, that name will no longer point to the buffer field; therefore you won't be able to access that field using the previous field name.

For example,

```
A$=B$
```

nullifies the effect of the FIELD statement above (line 100).

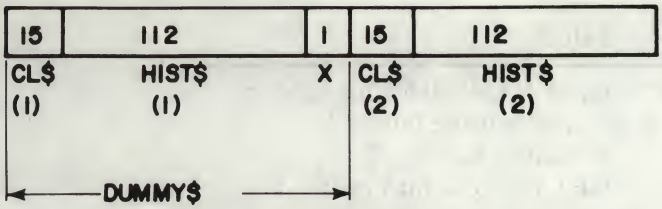
During random input, the GET statement places data into the 255-byte buffer, where it can be accessed using the field names assigned to that buffer. During random output, LSET and RSET place data into the buffer, so you can then PUT the buffer contents into a disk file.

Often you'll want to use a dummy variable in a FIELD statement to "pass over" a portion of the buffer and start fielding it somewhere in the middle. For example:

```
FIELD 1,15 AS CLIENT$(1),112 AS HIST$(1)
FIELD 1,128 AS DUMMY$,15 AS CLIENT$(2),112 AS HIST$(2)
```

In the second FIELD statement, DUMMY$ serves to move the starting position of CLIENT$(2) to position 129. In this manner, two identical "subrecords" are defined on buffer number 1. We won't actually use DUMMY$ to place data into the buffer or retrieve it from the buffer.

The buffer now "looks" like this:



Note that only one byte (the 128th byte) is left unused in this field structure.

GET
(read a record from disk – random access)

GET *nmexp1*[,*nmexp2*]

where *nmmexp1* specifies a random access file buffer, *nmexp1*=1,2, . . . ,15
nmexp2 specifies which record to GET in the file; if omitted, the current record will be read.

This statement gets a data record from a disk file and places it in the specified buffer. Before GETting data from a file, you must open the file and assign a buffer to it. That is, a statement like:

OPEN "R",*nmexp1*,*filespec*

is required **before** the statement:

GET *nmexp1*,*nmexp2*

When BASIC encounters the GET statement, it looks at the buffer's control block, and obtains:

- the information needed to access the file
- the mode in which this buffer was set up (must be R)
- the current record number
- The EOF (end-of-file) record number, i.e., the highest numbered record in the file
- lots of other information for internal use

BASIC then reads record *nmexp2* from the file and places it into the buffer. If you omit the record number, it will read the current record.

The "current record" is the record whose number is one higher than that of the last record accessed. The first time you access a file via a particular buffer, the current record is set equal to 1.

For example:

Program statement	Effect
1000 OPEN"R",1,"NAME/BAS"	Open NAME/BAS for random access using buffer 1
1010 FIELD 1,...	Structure buffer
1020 GET 1	GET record 1 into buffer 1
1025 REM... ACCESS BUFFER	
1030 GET 1,30	GET record 30 into buffer 1
1035 REM... ACCESS BUFFER	
1040 GET 1,25	GET record 25 into buffer 1
1046 REM... ACCESS BUFFER	
1050 GET 1	GET record 26 into buffer 1

If you attempt to GET a record whose number is higher than that of the end-of-file record, BASIC will fill the buffer with hex zeroes, and no error will occur.

To prevent this from occurring, you can use the LOF function to determine the number of the highest numbered record.

PUT
(write a record to disk – random access)

PUT *nmexp1*[,*nmexp2*]

where *nmexp1* specifies a random access file buffer, *nmexp*=1,2,...,15

nmexp2 specifies the record number in the file, *nmexp2*=1,2,.., up to 335, depending on how much space is available on the disk; if *nmexp2* is omitted, the current record number is assumed.

This statement moves data from a file's buffer into a specified place in the file. Before PUTing data in a file, you must:

1) OPEN the file, thereby assigning a buffer and defining the access mode (must be R);
2) FIELD the buffer, so you can
3) place data into the buffer with LSET and RSET statements.

When BASIC encounters the statement:

PUT *nmexp,nmexp2*

it does the following:

- Gets the information needed to access the disk file
- Checks the access mode for this buffer (must be R)
- Acquires more disk space for the file if necessary to accommodate the record indicated by *nmexp2*
- Copies the buffer contents into the specified record of the disk file
- Updates the current record number to equal *nmexp2+1*

The "current record" is the record whose number is one higher than the last record accessed. The first time you access a file via a particular buffer, the current record is set equal to 1.

If the record number you PUT is higher than the end-of-file record number, then *nmexp2* becomes the new end-of-file record number.

This has an important implication. When you PUT a record whose number exceeds the EOF record number, space is allocated on the disk to accommodate the new highest record number plus all lower-numbered records. For example,

PUT *nmexp*,336

will always produce a DISK FULL message, since TRSDOS attempts to find space for all records from 1 to 336 – and 335 is the maximum number of records available on a diskette.

Examples (assume a file named SAMPLE/BAS exists and that you have previously written 10 records to it, so that LOF=10):

Program statement	Effect
1000 OPEN"R",1,"SAMPLE/BAS"	Open SAMPLE/BAS for random address under buffer 1
1010 FIELD 1,.....	Prepare buffer
1020 LSET	Place data in buffer
1030 PUT 1	Copy buffer contents into current record (=#1)
1035 LSET	Place data in buffer
1040 PUT 1,30	Acquire disk space for records 2 through 30 and copy buffer contents into record 30; set LOF=30
1045 LSET	Place data in buffer
1050 PUT 1,25	Copy buffer contents into record 25
1055 LSET	Place data in buffer
1060 PUT 1	Copy buffer contents into current record (=#26)

LSET and RSET
(place data in a random buffer field)

LSET *var$* = *exp$* and RSET *var$* = *exp$*

where *var$* is a field name

exp$* contains the data to be placed in the buffer field named by *var$

These two statements let you place character-string data into fields previously set up by a FIELD statement.

For example, suppose NM$ and AD$ have been defined as field names for a random file buffer. NM$ has a length of 18 characters, and AD$ has a length of 25 characters.

Now we want to place the following information into the buffer fields so it can be written to disk:

name: JIM CRICKET, JR.
address: 2000 EAST PECAN ST.

This is accomplished with the two statements:

```
LSET NM$="JIM CRICKET, JR. "
LSET AD$="2000 EAST PECAN ST. "
```

This puts the data in the buffer as follows:

JIM␣CRICKET,JR.␣␣␣	2000␣EAST␣PECAN␣ST.␣␣␣␣␣␣
NM$	**AD$**

Note that filler spaces were placed to the right of the data strings in both cases. If we had used RSET instead of LSET statements, the filler spaces would have been placed on the left. This is the **only** difference between LSET and RSET.

For example:

```
RSET NM$="JIM CRICKET, JR. "
RSET AD$="2000 EAST PECAN ST. "
```

places data in the fields as follows:

␣␣␣JIM␣CRICKET,JR.	␣␣␣␣␣␣2000␣EAST␣PECAN␣ST.
NM$	**AD$**

If a string item is too large to fit in the specified buffer field, it is always truncated on the right. That is, the extra characters on the right are ignored.

CVD, CVI and CVS (restore string to numeric form)

CVD(*exp$*)

where *exp$* defines an eight character string; *exp$* is typically the name of a buffer-field containing a numeric string. If LEN(*exp$*)<8, an ILLEGAL FUNCTION CALL error occurs; if LEN(*exp$*)>8, only the first eight characters are used.

CVI(*exp$*)

where *exp$* defines a two-character string; *exp$* is typically the name of a buffer-field containing a numeric string. If LEN(*exp$*)<2, an ILLEGAL FUNCTION CALL error occurs; if LEN(*exp$*)>2, only the first two characters are used.

CVS(*exp$*)

where *exp$* defines a four-character string; *exp$* is typically the name of a buffer-field containing a numeric string. If LEN(*exp$*)<4, an ILLEGAL FUNCTION CALL error occurs; if LEN(*exp$*)>4, only the first four characters are used.

These functions let you restore data to numeric form after it is read from disk. Typically the data has been read by a GET statement, and is stored in a random access file buffer.

The functions CVD, CVI, CVS are inverses of MKD$, MKI$, and MKS$, respectively.

For example, suppose the name GROSSPAY$ references an eight-byte field in a random-access file buffer, and after GETting a record, GROSSPAY$ contains a MKD$ representation of the number 13123.38.

Then the statement:

```
PRINT CVD(GROSSPAY$)-TAXES
```

prints the result of the difference, 13123.38–TAXES. Whereas the statement:

```
PRINT GROSSPAY$-TAXES
```

will produce a TYPE MISMATCH error, since string values cannot be used in arithmetic expressions.

Using the same example, the statement

```
A#=CVD(GROSSPAY$)
```

assigns the numeric value 13123.38 to the double-precision variable A#.

EOF (end-of-file detector)

EOF(*nmexp*)

where *nmexp* specifies a file buffer,
nmexp=1,2,...,15

This function checks to see whether all characters up to the end-of-file marker have been accessed, so you can avoid INPUT PAST END errors during sequential input.

Assuming *nmexp* specifies an open file, then EOF(*nmexp*) returns 0 (false) when the EOF record has not yet been read, and –1 (true) when it has been read.

Examples:

```
IF EOF(5) THEN PRINT"END OF FILE"FILENM$
IF EOF(NM%) THEN CLOSE NM%
```

The following sequence of lines reads numeric data from DATA/TXT into the array A( ). When the last data character in the file is read, the EOF test in line 30 "passes", so the program branches out of the disk access loop, preventing an INPUT PAST END error from occurring. Also note that the variable I contains the number of elements input into array A( ).

```
5 DIM A(100) 'ASSUMING THIS IS A SAFE VALUE
10 OPEN "I",1,"DATA/TXT"
20 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1,A(I%)
50 I%=I%+1
60 GOTO 30
70 REM PROGRAM CONTINUES HERE AFTER DISK INPUT
```

LOF (get end-of-file record number)

LOF(*nmexp*)

where *nmexp* specifies a random access buffer
nmexp=1,2,...,15

This function tells you the number of the last, i.e., highest numbered, record in a file. It is useful for both sequential and random access.

For example, during random access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

Examples:

```
10 OPEN "R",1,"UNKNOWN/TXT"
20 FIELD 1,255 AS A$
30 FORI%=1 TO LOF(1)
40 GET 1,I%
50 PRINT A$
60 NEXT
```

In line 30, LOF(1) specifies the highest record number to be accessed.

Note: If you attempt to GET record numbers beyond the end-of-file record, BASIC simply fills the buffer with hexadecimal zeroes, and no error is generated.

When you want to add to the end of a file, LOF tells you where to start adding:

```
100 I%=LOF(1)+1 'HIGHEST EXISTING RECORD
110 PUT 1,I%    'ADD NEXT RECORD
```

MKD$, MKI$ and MKS$ (convert data, numeric-to-string)

MKD$(*nmexp*)

where *nmexp* is evaluated as a double-precision number

MKI$(*nmexp*)

where *nmexp* is evaluated as an integer, –32768<=*nmexp*<32768; if *nmexp* exceeds this range, an ILLEGAL FUNCTION CALL error occurs; any fractional component in *nmexp* is truncated

MKS$(*nmexp*)

where *nmexp* is evaluated as a single-precision number

These functions change a number to a "string". Actually the byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable. (See **LEVEL II Reference Manual**, VARPTR Function, for details of internal number representation.)

That is:

MKD$ returns an eight-byte string
MKI$ returns a two-byte string
MKS$ returns a four-byte string

Examples:

```
ASC(MKI$(I%)) equals the lsb of I%, i.e., (I% AND 255)
ASC(RIGHT$(MKI$(I),1))=the msb of I%, i.e., INT(I%/256)

LSET AVG$=MKS$(0.123)
```

AVG$ would typically reference a four-byte random buffer field. Now it contains a representation of the single-precision number 0.123.

```
LSET TALLY$=MKI$(I%)
```

Field name TALLY$ would now contain a two-byte representation of the integer I%.

```
A$=MKI$(8/I)
```

A$ becomes a two-byte representation of the integer portion of 8/I. Any fractional portion is ignored. Note that A$ in this case is a normal string variable, not a buffer-field name.

Suppose BASEBALL/BAT (a non-standard file extension) has been opened for random access using buffer 2, and the buffer has been FIELDed as follows:

field:	NM$	YRS$	AVG$	HR$	AB$	ERNING$
length:	16	2	4	2	4	4

NM$ is intended to hold a character string; AVG$, AB$ and ERNING$, converted single-precision values; YR$ and HR$, converted integers.

Suppose we want to write the following data record:

SLOW LEARNER played 38 years ; lifetime batting average .123; career homeruns, 11; at bats, 32768; . . . , earnings –13.75.

Then we'd use the make-string functions as follows:

```
1000 LSET NM$="SLOW LEARNER"
1010 LSET YRS$=MKI$(38)
1020 LSET AVG$=MKS$(.123)
1030 LSET HR$=MKI$(11)
1040 LSET AB$=MKS$(32768)
1050 LSET ERNING$=MKS$(-13.75)
```

After this sequence, you can write SLOW LEARNER's information to disk with the PUT statement. When you read it back from disk with GET, you will need to restore the numeric data from string to numeric form, using CVI and CVS functions.

Sequential Access Techniques

Sequential input/output is the simplest way to store data in disk files and retrieve it into BASIC variables.

To write to disk, you open a file for sequential output, PRINT# the data, and close the file. To read the data back, you simply open the file for sequential access and INPUT# the data directly into BASIC variables – in the same order as the data was written onto the disk.

Sequential Output – An Example

Suppose we want to store a table of English-to-metric conversion constants:

English unit	Metric equivalent
1 inch	2.54001 centimeters
1 mile	1.60935 kilometers
1 acre	4046.86 sq. meters
1 cubic inch	0.01638716 liter
1 U.S. gallon	3.785 liters
1 liquid quart	0.9463 liter
1 lb (avoir)	0.45359 kilogram

First we decide what the data image is going to be. Let's say we want it to look like this:

english unit–>*metric unit*, *factor* <EN>

For example, the stored data would start out:

IN–>CM,␢2.54001␢ <EN>

The following program will create such a data file.

Note: <EN> represents a carriage return, hex OD.

```
10 OPEN"O",1,"METRIC/TXT"
20 FORI%=1 TO 7
30 READ UNIT$,FACTR
40 PRINT#1,UNIT$;",";FACTR
50 NEXT
60 CLOSE
70 DATA IN->CM,2.54001,MI->KM,1.60935,ACRE->SQ.M,4046.86
80 DATA CU.IN->LTR,1.638716E-2,GAL->LTR,3.785
90 DATA LIQ.QT->LTR,0.9463,LB->KG,0.45359
```

Line 10 creates a disk file named METRIC/TXT, and assigns buffer 1 for sequential output to that file. The extension /TXT is used because sequential output always stores the data as ASCII-coded text.

Note: If METRIC/TXT already exists, line 10 will cause all its data to be lost. Here's why: Whenever a file is opened for sequential output, the EOF marker is set to the beginning of the file. In effect, TRSDOS "forgets" that anything has ever been written beyond this point.

Line 40 prints the current contents of UNIT$ and FACTR to the file buffer. The disk-write won't actually take place until the buffer is filled or you close the file, whichever happens first. Since the string items do not contain delimiters, it is not necessary to print explicit quotes around them. The explicit comma is sufficient.

Line 60 closes the file. The EOF marker points to the end of the last data item, i.e., 0.45359, so that later, during input, DISK BASIC will know when it has read all the data.

Sequential Input – An Example

The following program reads the data from METRIC/TXT into two "parallel" arrays, then asks you to enter a conversion problem.

```
5 CLEAR 500
10 DIM UNITS$(9),FACTR(9) 'ALLOWS FOR UP TO 10 DATA PAIRS
20 OPEN"I",1,"METRIC/TXT"
25 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1,UNIT$(I%),FACTR(I%)
50 I%=I%+1
60 GOTO 30
70 REM...THE CONVERSION FACTORS HAVE BEEN READ IN
100 CLS: PRINT TAB(5)"*** ENGLISH TO METRIC CONVERSIONS ***"
110 FOR ITEM%=0TOI%-1
120 PRINT USING"(## )          %        %";ITEM%,UNIT$(ITEM%)
130 NEXT
140  PRINT@704,"WHICH CONVERSION ";
150 INPUT CHOICE%
155 PRINT@768,"ENTER ENGLISH QUANTITY";
160 INPUT V
170 PRINT"THE METRIC EQUIVALENT IS"V*FACTR(CHOICE%)
180 INPUT"PRESS ENTER TO CONTINUE";X
190 PRINT@704,CHR$(31); 'CLEAR TO END OF FRAME
200 GOTO 140
```

Line 20 opens the file for sequential input. The read pointer is automatically set to the beginning of the file.

Line 30 checks to see that the end-of-file record hasn't been read. If it has, control branches from the disk input loop to the part of the program that uses the newly acquired data.

Line 40 reads a value into the string array UNIT$( ), and a number into the single-precision array FACTR( ). Note that this INPUT list parallels the PRINT# list that created the data file (see the section "Sequential Output: An example"). This parallelism is not required, however. We could just as successfully have used:

```
40 INPUT#1,UNIT$(I%): INPUT#1,FACTR(I%)
```

How to update a file

Suppose you want to add more entries into the English-Metric conversion file. You can't simply re-open the file for sequential output and PRINT# the extra data – that would immediately set the end-of-file marker to the beginning of the file, effectively destroying the file's previous contents. Do this instead:

1) Open the file for sequential input
2) Input the entire file and store it (typically in one or more arrays)
3) Close the file
4) Add your new entries to the data array, or correct existing entries
5) Re-open the file for sequential output
6) Output the updated data array to the file
7) Close the file

If the file is too large to fit in memory, update it this way:

1) Open the file for sequential input
2) Open another new data file for sequential output
3) Input a block of data and update the data as necessary
4) Output the data to the new file
5) Repeat steps 3 and 4 until all data has been read, updated, and output to the new file; then go to step 6
6) Close both files
7) Kill the old data file
8) Rename the new file (TRSDOS RENAME command) to the name of the old file.

Sequential LINE INPUT - An Example

Using the line-oriented input, you can write programs that edit other BASIC program files : renumber them, change LPRINTs to PRINTs, etc. – as long as these "target" programs are stored in ASCII format.

The following program counts the number of lines in any disk file with the extension "/TXT".

```
10 CLEAR 300
20 INPUT "WHAT IS THE NAME OF THE PROGRAM"; PROG$
30 IF INSTR(PROG$,"/TXT")=0 THEN 110 'REQUIRE /TXT EXTENSION
40 OPEN"I",1,PROG$
50 I%=0
60 IF EOF(1)THEN 90
70 I%=I%+1: LINE INPUT#1,TEMP$
80 GOTO60
90 PRINT"THE PROGRAM IS"I%"LINES LONG."
100 CLOSE: GOTO20
110 PRINT "FILESPEC MUST INCLUDE THE EXTENSION '/TXT'"
120 GOTO20
```

For BASIC programs stored in ASCII, each program line ends with an < EN > character not preceded by an < LF > line feed.
So the LINE INPUT in line 70 automatically reads one entire line at a time, into the variable TEMP$. Variable I% actually does the counting.

To try out the program, save DISKDUMP/BAS as a text file:

```
LOAD"DISKDUMP/BAS"  ENTER
SAVE"DISKDUMP/TXT",A ENTER
```

This gives you a second, ASCII-format version of DISKDUMP.

Now type in the line-counter program and tell it to examine the program DISKDUMP/TXT.

Disk Storage during Sequential Access

One thing that makes sequential access so simple is that you can generally ignore the details of disk storage. You just write your data and read it back.

Described below are a few of the technical details and hints you should keep in the back of your mind. In some situations, they will become important.

1. PRINT# statements don't write data directly to the disk; instead, the data is placed in the 256-byte output buffer. When this buffer is filled, the contents are automatically written to disk. (Closing the file will also write the buffer to disk.)

2. If a DISK FULL ERROR occurs during execution of a PRINT# statement, you should realize that the current contents of the output buffer have not been written to the file. The data in the disk file is intact, but it doesn't contain the last few values you PRINTed to it.

   If your variables still contain the data, you can recover it directly.

Random Access Techniques

Random access offers several advantages over sequential access:

- Instead of having to start reading at the beginning of a file, you can read any record you specify.
- To update a file, you don't have to read in the entire file, update the data, and write it out again. You can rewrite or add to any record you choose, without having to go through any of the other records.
- Random access is more efficient – data takes up less space and is read and written faster.
- Opening a file for random access allows you to write to and read from the file via the same buffer.
- Random access provides many powerful statements and functions to structure your data. Once you have set up the structure, random input/output becomes quite simple.

The last advantage listed above is also the "hard part" of random access. It takes a little extra thought.

For the purposes of random access, you can think of a disk file as a set of boxes – like a wall of post-office boxes. Just like the post office receptacles, the file boxes are numbered.

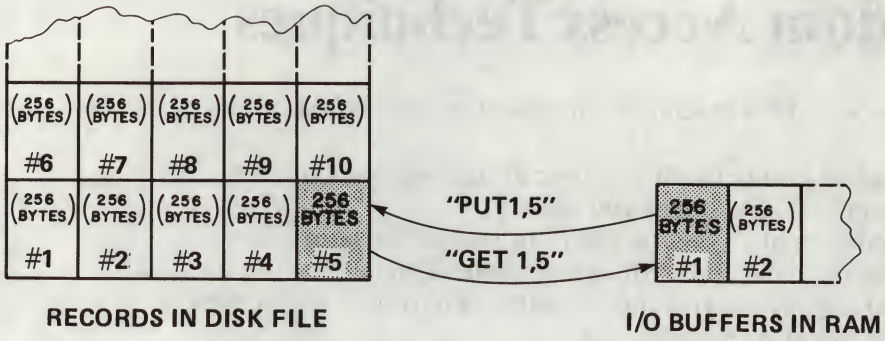
The number of boxes in a file will vary, but it's always a multiple of 5.

The smallest non-empty file contains 5 boxes, numbered 1 through 5. When the file needs more space to hold more data, TRSDOS provides it in increments of 5.

These fixed-sized boxes are referred to as "records". Each record contains 256 bytes, 255 of which are available for storing your data.

You can place data in any record, or read the contents of any record, with statements like:

```
PUT 1,5     write buffer-1 contents to record 5
GET 1,5     read the contents of record 5 into buffer-1
```



The buffer is a waiting area for the data. Before writing data to a file, you must place it in the buffer assigned to the file. After reading data from a file, you must retrieve it from the buffer.

As you can see from the sample PUT and GET statements above, data is passed to and from the disk in 256-byte chunks.

"That's a lot of data." But most values occupy only a few bytes:

Integers	2
Single-precision numbers	4
Double precision numbers	8
Strings	Up to 255

Therefore you'll want to place several values into the buffer before PUTting its contents into the disk file, to avoid wasting disk space.

This is accomplished by 1) dividing the buffer up into fields and naming them, then 2) placing the string or numeric data into the fields.

For example, suppose we want to store a glossary on disk. Each record will consist of a word followed by its definition. We start with:

```
100 OPEN"R",1,"GLOSSARY/BAS"
110 FIELD 1,15 AS WD$,240 AS MEANING$
```

Line 100 opens a file named GLOSSARY/BAS (creates it if it doesn't already exist); and gives buffer 1 random access to the file.

Line 110 defines two fields onto buffer 1:
WD$ consists of the first 15 bytes of the buffer;
MEANING$ consists of the last 240 bytes.

WD$ and MEANING$ are now **field-names.**

What makes field names different? Most string variables point to an area in memory called the string space. This is where the value of the string is stored.

Field names, on the other hand, point to the buffer area assigned in the FIELD statement. So, for example, the statement:

```
10 PRINT WD$ ": " MEANING$
```

displays the contents of the two buffer fields defined above.

These values are meaningless unless we first place data in the buffer. LSET, RSET and GET can all be used to accomplish this function. We'll start with LSET and RSET, which are used in preparation for disk output.

Our first entry is the word "left-justify" followed by its definition.

```
100 OPEN"R",1,"GLOSSARY/BAS"
110 FIELD 1,15 AS WD$,240 AS MEANING$
120 LSET WD$="LEFT-JUSTIFY"
130 LSET MEANING$="TO PLACE A VALUE IN A FIELD FROM LEFT
TO RIGHT; IF THE DATA DOESN'T FILL THE FIELD, BLANKS ARE ADDED
ON THE RIGHT; IF THE DATA IS TOO LONG, THE EXTRA CHARACTERS ON
THE RIGHT ARE IGNORED. LSET IS A LEFT-JUSTIFY FUNCTION."
```

Line 120 left-justifies the value in quotes into the first field in buffer 1. Line 130 does the same thing to its quoted string. When typing in line 130, you should insert line-feed <LF> characters (press the down arrow) to force line breaks as above. This makes it easier to print out the data after reading it back in to a string variable.

Note: RSET would place filler-blanks to the **left** of the item. Truncation would still be on the right.

Now that the data is in the buffer, we can write it to disk with a simple PUT statement:

```
140 PUT 1,1
150 CLOSE
```

This writes the first record into the file GLOSSARY/BAS.

To read and print the first record in GLOSSARY/BAS, use the following sequence:

```
160 OPEN"R",1,"GLOSSARY/BAS"
170 FIELD 1, 15 AS WD$,240 AS MEANING$
180 GET 1,1
190 PRINT WD$ ":  " MEANING$
200 CLOSE
```

Lines 160 and 170 are required only because we closed the file in line 150. If we hadn't closed it, we could go directly to line 180.

Random Access: A general procedure

The above example shows the necessary sequences to read and write using random access. But it does not demonstrate the primary advantages of this form of access – in particular, it doesn't show how to update existing files by going directly to the desired record.

The program below, GLOSSACC/BAS, develops the glossary example to show some of the techniques of random access for file maintenance. But before looking at the program, study this general procedure for creating and maintaining files via random access.

Step Number	See GLOSSACC/BAS, Line Number
1. OPEN the file	110
2. FIELD the buffer	120
3. GET the record to be updated	140
4. Display current contents of the record (use CVD,CVI,CVS before displaying numeric data)	145-170
5. LSET and RSET new values into the fields (use MKD$,MKI$,MKS$ with numeric data before setting it into the buffer)	210-230
6. PUT the updated record	240
7. To update another record, continue at step 3. Otherwise, go to step 8.	250-260
8. Close the file	270

```
10 REM. . GLOSSACC/BAS. . .
100 CLS: CLEAR 300
110 OPEN"R",1,"GLOSSARY/BAS"
120 FIELD 1,25 AS WD$,228 AS MEANING$,2 AS NX$
130 INPUT"WHAT RECORD DO YOU WANT TO ACCESS";R%
140 GET 1,R%
145 NX%=CVI(NX$)   'SAVE LINK TO NEXT ALPHABETICAL ENTRY
150 PRINT"WORD:  "WD$
160 PRINT"DEF'N:": PRINTMEANING$
170 PRINT"NEXT ALPHABETICAL ENTRY: RECORD#"NX%: PRINT
180 W$="": INPUT"TYPE NEW WORD<EN> OR <EN> IF OK";W$
190 D$="":PRINT"TYPE NEW DEF'N<EN> OR <EN> IF OK?":LINEINPUTD$
200 INPUT"TYPE NEW SEQUENCE NUMBER OR <EN> IF OK";NX%
210 IF W$<>""THEN LSET WD$=W$
220 IF D$<>""THEN LSET MEANING$=D$
230 LSET NX$=MKI$(NX%)
240 PUT 1,R%
245 R%=NX%  'USE NEXT ALPHA. LINK AS DEFAULT FOR NEXT RECORD
250 CLS: INPUT"    TYPE<EN> TO READ NEXT ALPHA. ENTRY,
    OR RECORD # <EN> FOR SPECIFIC ENTRY,
    OR 0 <EN> TO QUIT";R%
260 IF 0<R% THEN  140
270 CLOSE
280 END
```

Notice we've added a field, NX$, to the record (line 120). NX$ will contain the number of the record which comes next in alphabetical sequence. This enables us to proceed alphabetically through the glossary, provided we know which record contains the entry which should come first.

For example, suppose the glossary contains:

record#	word (WD$)	defn,	pointer to next alpha. entry (NX$)
1	LEFT-JUSTIFY	. . .	3
2	BYTE	. . .	4
3	RIGHT-JUSTIFY	. . .	0
4	HEXADECIMAL	. . .	1

When we read record 2 (BYTE), it tells us that record 4 (HEXADECIMAL) is next, which then tells us record 1 (LEFT-JUSTIFY) is next, etc. The last entry, record 3 (RIGHT-JUSTIFY), points us to zero, which we take to mean "THE END".

Since NX$ will contain an integer, we have to first convert that number to a two-byte string representation, using MKI$ (line 230 above).

```
RUN"GLOSSACC/BAS" ENTER
```

```
WHAT RECORD DO YOU WANT TO ACCESS? 4 ENTER
WORD:  HEXIDECIMAL
DEF'N:
CAPABLE OF EXISTING IN ANY OF 16 STATES, E.G., THE  HEXADECIMAL
DIGITS 0,1,2,...,9,A,B,C,D,E,F.  HEXADECIMAL NUMBERS ARE STRINGS
OF HEXADECIMAL DIGITS.

NEXT ALPHABETICAL ENTRY: RECORD# 1

TYPE NEW WORD<EN> OR <EN> IF OK? HEXADECIMAL ENTER
TYPE NEW DEF'N<EN> OR <EN> IF OK? ENTER

TYPE NEW SEQUENCE NUMBER OR <EN> IF OK? ENTER
```

```
    TYPE<EN> TO READ NEXT ALPHA. ENTRY,
    OR RECORD # <EN> FOR SPECIFIC ENTRY,
    OR 0 <EN> TO QUIT? ENTER
WORD:  LEFT-JUSTIFY
DEF'N:
TO PLACE DATA IN A FIELD FROM LEFT TO RIGHT, ADDING BLANKS AS
NECESSARY ON THE RIGHT TO FILL THE FIELD.  ANY EXTRA CHARACTERS
ON THE RIGHT ARE IGNORED.

NEXT ALPHABETICAL ENTRY: RECORD# 3

TYPE NEW WORD<EN> OR <EN> IF OK? ENTER
TYPE NEW DEF'N<EN> OR <EN> IF OK? ENTER

TYPE NEW SEQUENCE NUMBER OR <EN> IF OK? 2 ENTER
```

The following program displays the glossary in alphabetical sequence:

```
300 REM. . GLOSSOUT/BAS. . .
310 CLS: CLEAR 300
320 OPEN"R",1,"GLOSSARY/BAS"
330 FIELD 1,15 AS WD$,238 AS MEANING$,2 AS NX$
340 INPUT"WHICH RECORD IS FIRST ALPHABETICALLY";N%
350 GET 1,N%
360 PRINT:PRINTWD$
370 PRINTMEANING$
380 N%=CVI(NX$)
390 INPUT"PRESS ENTER TO CONTINUE";X
400 IF N%<>0 THEN 350
410 CLOSE
420 END
```

Sub-Records

In the glossary example, each entry required the full 255 bytes available in the buffer. Often this is not the case. When each information-unit fills only a part of the buffer, it is a good idea to define several identical sub-records on the buffer. That way you don't waste disk space by PUTting records which contain only a few bytes of useful information.

For example, suppose we want to store a mailing list, and each entry will consist of:

field	field length
name	18
address	25
city	14
state	2
last purchase amt.	4

Total length of entry: 63

Note: The last-purchase-amount will be a single-precision number. Such values require 4 bytes, therefore the field length is 4.

If we didn't care about wasting space on the disk, we could use the following statement:

```
FIELD 1, 18 AS NM$,25 AS AD$,14 AS CTY$,2 AS ST$,4 AS LP$
```

PUTting such a buffer would create a record consisting of 63 bytes of information followed by 255-63=192 unused bytes.

A more efficient approach fields the buffer into identical sub-records. In this case, we can create 255/63 = 4 sub-records plus only 3 wasted bytes at the end.

Instead of using a very long FIELD statement to explicitly assign each field, we re-field the buffer once for each sub-record, using a dummy string, STARTHERE$, to start each sub-record at the appropriate position in the buffer.

```
FORI%=0 TO 3
      FIELD 1,(I%*63) AS STARTHERE$,18 AS NM$(I%),
      25 AS AD$(I%),14 AS CTY$(I%),2 AS ST$(I%),4 AS LP$(I%)
NEXT
```

The first time through the loop, STARTHERE$ will have a length of zero. Therefore NM$(0) will start at the first byte; AD$(0), at the 19th byte, etc.; LP$(0) will end at the 63rd byte.

The second time through the loop, STARTHERE$ will have a length of 63. Therefore NM$(1) will start at the 64th byte; AD$(1), at the 92nd byte, etc.; LPS(1) will end at the 126th byte.

And so forth, until the buffer is completely defined.

To place values in the subrecords of the buffer: assume our mailing list entries are stored in four arrays, N$( ),A$( ),C$( ),S$( ),LP( ).

Then we can fill the buffer with four entries as follows:

```
FOR I%=0TO3
       LSET NM$(I%)=N$(I%)
       LSET AD$(I%)=A$(I%)
       LSET CT$(I%)=C$(I%)
       LSET ST$(I%)=S$(I%)
       LSET LP$(I%)=MKS$(LP(I%))
NEXT
```

How to Access Sub-Records

Since each record in such a file will contain four sub-records, we need a way to pull out the sub-record we want. This requires that each sub-record have a unique number which can be related to the record which contains it.

For this example, suppose we have a printout of the entire mailing list, starting from the first sub-record in record 1 and going through to the last sub-record in the last record. We then number them sequentially, **starting with 1.**

The following formulas use this number (we'll call it a key-number) to determine exactly where the sub-record is in the file:

If the sub-record's key-number is KEY%, then

PR% = INT((KEY%–1)/4)+1

where PR% is the physical record that contains the sub-record, and

SR%= KEY% – 4*(PR%–1)

where SR% is the sub-record number inside the physical record. For example, suppose we want to access the entry with key number = 37 (i.e., the 37th entry). Then the physical record which contains it is:

INT((37–1)/4)+1 == > record 10

And its position in record 10 is:

37 – 4*(10–1)+1 == > sub-record number 1

A full working program for creating and manipulating a mailing list follows:

```
100 CLEAR 1000
110 OPEN"R",1,"MAIL/BAS"
120 CLS: INPUT"TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT";N%
130 IF N%=0 THEN CLOSE: END
140 INPUT"TYPE KEY NUMBER<EN> OR 0<EN>"; KEY%
150 IF KEY%=0 THEN 120
160 PR%=INT((KEY%-1)/4)+1
170 SR%=KEY%-4*(PR%-1)
180 FIELD 1,((SR%-1)*63) AS STARTHERE$,18 AS NM$, 25 AS AD$,
    14 AS CTY$,2 AS ST$,4 AS LP$
190 GET 1,PR%
200 IF N%=2THEN300
210 PRINT"WRITING SUBRECORD #"SR%"IN PHYSICAL RECORD #"PR%
220 PRINT: PRINT"NAME?"TAB(20);: LINEINPUT N$: LSET NM$=N$
230 PRINT"ADDRESS?"TAB(20);: LINEINPUT A$: LSET AD$=A$
240 PRINT"CITY?"TAB(20);: LINEINPUT C$: LSET CTY$=C$
250 PRINT"STATE?"TAB(20);: LINEINPUT S$: LSET ST$=S$
260 PRINT"LAST PURCHASE"TAB(20);:INPUTLP:LSEKS$(LP)
270 PUT 1,PR%: PRINT: INPUT"PRESS <EN> TO GO ON";X: GOTO 120
300 PRINT"READING SUBRECORD #"SR%"IN PHYSICAL RECORD #"PR%
310 PRINT: PRINT"NAME"TAB(20)NM$
320 PRINT"ADDRESS"TAB(20)AD$
330 PRINT"CITY"TAB(20)CTY$
340 PRINT"STATE"TAB(20)ST$
350 PRINT USING"LAST PURCHASE       $#####.##";CVS(LP$)
360 PRINT: INPUT"PRESS <EN> TO GO ON";X: GOTO120
```

This program actually doesn't require you to fill the buffer with four meaningful sub-records. As soon as you've placed a sub-record in the correct position in the field, the entire buffer is written to disk. However, the extra space is not wasted; it is always available for subsequent sub-records to be added.

Note that this would not be the most efficient way to create a list at one "sitting". In such a case you'd probably want to fill the buffer with four sub-records before doing the disk-write. The above program does, however, show you how to update a file using random access.

```
RUN"MAILACC/BAS" ENTER
```

```
TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT? 1 ENTER
TYPE KEY NUMBER<EN> OR 0<EN>? 3 ENTER
WRITING SUBRECORD # 3 IN PHYSICAL RECORD # 1

NAME?                DUNMAN, I.C. ENTER
ADDRESS?             2222 SECOND STREET ENTER
CITY?                OLD PORT ENTER
STATE?               AZ ENTER
LAST PURCHASE        ? 12.81 ENTER

PRESS <EN> TO GO ON? ENTER
```

```
TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT? 2 ENTER
TYPE KEY NUMBER<EN> OR 0<EN>? 1 ENTER
READING SUBRECORD # 1 IN PHYSICAL RECORD # 1

NAME                 JOHNSON, J.R.
ADDRESS              1024 RAM DRIVE
CITY                 FORT WUMPUS
STATE                TX
LAST PURCHASE        $ 188.75

PRESS <EN> TO GO ON? ENTER
```

```
TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT? 0 ENTER
READY
>_
```

Overlapping Fields

Suppose you want to access a field in two ways – in total and in part. Then you can assign two field names to the same area of the buffer.

For example, if the first two digits of a six-digit stock-number specify a category, you might use the following field structure:

```
FIELD 1, 6 AS STOCK$, .....
FIELD 1, 2 AS CTG$, .....
```

Now STOCK$ will reference the entire stock-number field, while CTG$ will reference only the first two digits of the number.

DISK BASIC Error Messages

Code	Message	Explanation
50	FIELD OVERFLOW	More than 255 bytes were allocated to a random-access buffer.
51	INTERNAL ERROR	Error in disk operating system itself, or disk I/O fault.
52	BAD FILE NUMBER	A file-buffer number was used improperly; number has not been assigned to a file with an OPEN statement.
53	FILE NOT FOUND	Attempt to read from a file which is not contained on the disk; check name/extension to see they were specified correctly.
54	BAD FILE MODE	Attempt to perform disk file input or output which conflicts with the mode in which the file was opened.
57	DISK I/O ERROR	An error occurred during data transfer between the Computer and a disk file.
61	DISK FULL	All available space on the diskette has been used.
62	INPUT PAST END	During sequential input to a variable, the end of file was reached before any data characters were read.
63	BAD RECORD NUMBER	Record number in a PUT statement exceeded the range <1,340>.
64	BAD FILENAME	An invalid file specification was provided; study "File Specification", **TRSDOS Overview**.

Note: Disk errors cannot be simulated via the ERROR statement

Code	Message	Explanation
66	DIRECT STATEMENT IN FILE	Attempt to LOAD, RUN, or MERGE a disk file which is not a BASIC program.
67	TOO MANY FILES	Attempt to place more than 48 files on a single diskette.
68	DISK WRITE-PROTECTED	Attempt to write to disk with write-protect notch covered.
69	FILE ACCESS DENIED	Attempt to access existing file with incorrect password.

Appendices



APPENDICES

Contents of This Section

Glossary 2
Memory Map 3
TRSDOS Character Tables 14
Base Conversions 18

Glossary

access

The method in which information is read from or written to disk; see random access and sequential access.

address

A location in memory, usually specified as a two-byte hexadecimal number. The address range<0 to FFFF>is represented in decimal as<0 to 32767> <-32768,...,-1>

alphabetic

Referring strictly to the letters A-Z.

alphanumeric

Referring to the set of letters A-Z and the numerals 0-9.

argument

The string or numeric quantity which is supplied to a function and is then operated on to derive a result; this result is referred to as the **value** of the function.

array

An organized set of elements which can be referenced in total or individually, using the array name and one or more subscripts. In BASIC, any variable name can be used to name an array; and arrays can have one or more dimensions. AR( ) signifies a one-dimensional array named AR; AR(,) signifies a two-dimensional array named AR; etc.

ASCII

American Standard Code for Information Interchange. This method of coding is used to store textual data. Numeric data is typically stored in a more compressed format.

ASCII format disk file

Disk files in which each byte corresponds to one character of the original data. For example, a BASIC program stored in ASCII format "looks like" the program listing, except that each character is ASCII-coded. Compare to compressed-format file.

background task

A relatively slow routine which the computer executes along with other background tasks, and which is subject to interrupts. When the interrupt-driven tasks are completed, the background task continues. See **foreground task, task.**

backup disk

An exact copy of the original: a "safe copy". You should keep backups of your original TRSDOS diskette and all important data diskettes.

BASIC

Beginners' All-purpose Symbolic Instruction Code, the programming language which is stored in ROM in the TRS-80. Radio Shack supports LEVEL I BASIC, LEVEL II BASIC, and DISK BASIC. LEVEL II is a subset of DISK BASIC.

baud

Signalling speed in bits per second. The LEVEL II cassette interface operates at 500 baud.

binary

Having two possible states, e.g., the binary digits 0 and 1. The binary (base 2) numbering system uses sequences of zeroes and ones to represent quantities. This is analagous to the Computer's internal representation of date, using electrical values for 0 and 1.

bit

Binary digit; the smallest unit of memory in the Computer, capable of representing the values 0 and 1.

bootstrap program

A fundamental or primitive program which takes the Computer from an OFF condition to one in which it is capable of loading and executing a higher-level program – i.e., a program which allows the Computer to pull itself up "by its own bootstraps". A program which initializes the Computer.

break

To interrupt execution of a program. In BASIC the statement

```
STOP
```

causes a break in execution, as does pressing the BREAK key.

buffer

An area in RAM where data is accumulated for further processing. For example, to pass data from BASIC to a disk file, and vice-versa, the data must go through a file-buffer.

buffer field

A portion of the buffer which you define as the storage area for a buffer-field variable. Dividing a buffer into fields allows you to pass multiple values to and from disk storage.

byte

The smallest addressable unit of memory in the Computer, consisting of 8 consecutive bits, and capable of representing 256 different values, e.g., decimal values from 0 to 255.

compressed-format

A method of storing information in less space than a standard ASCII representation would require. An integer always requires two bytes; a single-precision number, four; a double-precision number, 8 — regardless of how many characters are required to represent the numbers as text. String values cannot be stored in compressed format.

BASIC programs in RAM and non-ASCII disk files are stored in compressed-format, with all BASIC keywords stored as special one-byte codes.

command file

A TRSDOS disk file with the extension /CMD. Such a file should consist of an executable Z-80 program, since TRSDOS will load and attempt to execute it when you type:

filename **ENTER**

Command files can be placed on any disk; in effect, they extend the set of TRSDOS library commands (though, of course, they remain external to the TRSDOS system files).

close

Terminate access to a disk file. Before re-accessing the file, you must re-open it.

data

Information that is passed to our output from a program; under LEVEL II and DISK BASIC, there are four types of data:

- integer numbers
- single-precision floating point numbers
- double-precision floating point numbers
- character-string sequences, or just "strings"

data/device control block (DCB)

An area in RAM associated with an I/O buffer, containing information the Operating System requires in order to access the I/O device or file.

debug

To isolate and remove logical or syntax errors from a program.

decimal

Capable of assuming one of ten states, e.g., the decimal digits 0,1,...,9. Decimal (base 10) numbering is the everyday system, using sequences of decimal digits. Decimal numbers are stored in binary code in the Computer.

default

An action or value which is supplied by the Computer when you do not specify an action or value to be used.

delimiter

A character which marks the beginning or end of a data item, and is not a part of the data. For example, the double-quote symbol is a string delimiter to BASIC.

destination

The device or address which receives the data during a data transfer operation. For example, during a BACKUP operation, the destination disk is the one onto which the source-disk is being copied.

device

A physical part of the computer system used for data I/O, e.g., keyboard, display, line printer, cassette, disk drive, voice synthesizer.

directory

A listing of the files which are contained on a disk.

disk drive or **Mini Disk drive**

The physical device which writes data onto diskettes and retrieves it.

diskette or **disk**

A magnetic recording medium for mass data storage.

drive specification or **drivespec**

An optional field in a TRSDOS file specification and in some TRSDOS commands, consisting of a colon followed by one of the digits 0 through 3. The drivespec is used to specify which drive is to be used for a disk read or write.

When the drivespec is omitted from a command involving a read operation, TRSDOS will search through all the disks for the desired file, starting with drive 0.

When the drivespec is omitted from a command involving a write operation, TRSDOS will generally search through all non write-protected drives for the desired file.

drive number

An integer value from 0 to 3, specifying one of the Mini Disk drives. Drive 0 is closest to the Expansion Interface, and Drive 3 is farthest away. Drive 0 must always contain the TRSDOS diskette, with a couple of exceptions.

dummy variable

A variable name which is used in an expression to meet syntactic requirements, but whose value is insignificant to the programmer.

edit

To change existing information.

end of file or **EOF**

A marker which indicates the end of a disk file, i.e., where the meaningful data ends and the unknown begins.

entry point

The address of a machine-language program or routine where execution is to begin. This is not necessarily the same as the starting address. Entry point is also referred to as the **transfer address.**

expression

A meaningful sequence of one or more variables, constants, operators and functions.

field

A user-defined subdivision of a random access file-buffer, created and named with the FIELD statement.

field name

A string variable which has been assigned to a field in a random access file-buffer via the FIELD statement.

file

An organized collection of related data. Under TRSDOS, a file is the largest block of information which can be addressed with a single command. BASIC programs and data sets are stored on disk in distinct files.

file extension

An optional field in a file specification, consisting of a / followed by one alphabetic and up to two alphanumeric characters; the extension can be used to identify the file type, e.g., /BAS, /TXT, /CIM, for BASIC, text, and core image, respectively.

filename

A required field in a file specification, consisting of one alphabetic followed by up to 7 alphanumeric characters. Filenames are assigned when a file is created or renamed.

file specification or **filespec**

A sequence of characters which specifies a particular disk file under TRSDOS, consisting of a mandatory filename, followed by an optional extension, password, and drivespec.

foreground task

A relatively fast routine which the Computer must execute periodically, in sequence with other foreground tasks. Such tasks are interrupt-driven. See **background task, task, interrupt.**

format

To organize a new or magnetically erased diskette into tracks and sectors, via the TRSDOS FORMAT utility. BACKUP also implicitly formats a blank diskette. Formatted diskettes contain 35 tracks, each of which contains 10 sectors.

granule

The smallest unit of allocatable space on a disk, consisting of 5 sectors.

hexadecimal or **hex**

Capable of existing in one of 16 possible states. For example, the hexadecimal digits are 0,1,2,. . ,,9,A,B,C,D,E,F. Hexadecimal (base-16) numbers are sequences of hexadecimal digits. Address and byte values are frequently given in hexadecimal form. Under DISK BASIC, hexadecimal constants can be entered by prefixing the constant with &H.

increment

The value which is added to a counter each time one cycle of a repetitive procedure is completed.

input

To transfer data from outside the Computer (from a disk file, keyboard, etc.) into RAM.

interrupt

A signal which causes the Computer to interrupt whatever it is doing and perform some other specified task; when the task is completed, the Computer will generally resume execution of the previous task. The TRS-80 Expansion Interface includes a 25 millisecond "heartbeat" interrupt, which is used to drive the real-time clock and other foreground tasks. Interrupt-driven tasks can be scheduled and assigned priorities, so that the Computer appears to be doing two or more things "at once".

kilobyte or **K**

1024 bytes of memory. Thus a 12 K ROM includes 12*1024=12288 bytes.

library commands

A set of overlayed TRSDOS commands which are overlayed as needed into RAM between 5200 and 6FFF, to see which library commands are available, use the TRSDOS LIB command:

```
LIB <EN>
```

logical expression

An expression which is evaluated as either True (=-1) or FALSE (=0).

logical record

A block of data which contains from 1 to 256 bytes, and can be addressed as a unit, regardless of whether the logical record is contained in a single record or spans two physical records.

machine language

The Z-80 instruction set, usually specified in hexadecimal code. All higher-level languages must be translated into machine-language in order to be executed by the Computer.

null string

A string which has a length of zero; For example, the assignment

```
A$ = " "
```

makes A$ a null-string.

object code

Machine language derived from "source code", typically, from Assembly Language.

octal

Capable of existing in one of 8 states, for example, the octal digits are 0,1,...,7. Octal (base-8) numbers are sequences of octal digits. Address and byte values are frequently given in octal form. Under DISK BASIC, an octal constant can be entered by prefixing the octal number with the symbol &O.

open

To prepare a file for access by assigning a sequential input, sequential output, or random I/O buffer to it.

output

To transfer data from inside a Computer's memory to some external area, e.g., a disk file or a line printer.

overlay

To replace one block of code in RAM with another block. Also, the code which replaces the previous contents of RAM. For example, the TRSDOS system routines are stored on disk and loaded into a common area of RAM as **overlays.**

parameter

Optional information supplied with a command to specify how the command is to operate. TRSDOS parameters are placed inside parentheses.

password

An optional field in a filespec consisting of one alphanumeric followed by up to 7 additional alphanumeric characters. If a file is created without a password, 8 blanks become the default password. To access a file, you must specify the password in the filespec.

Using the TRSDOS ATTRIB command, you can assign both update and access passwords; the access password will grant only a limited degree of access, while the update password grants total access to the file. See **filespec.**

physical record

The smallest amount of data which can be written to a disk file or read from it; under TRSDOS, physical records consist of 256 bytes. Note that physical record length can be ignored by the assembly-language programmer, since TRSDOS supports logical records of from 1 to 256 bytes in length.

prompt

A character or message provided by the Computer to indicate that it's ready to accept keyboard input.

protected file

A disk file which has a non-blank password, and therefore can only be accessed by reference to that password.

protection level

The degree of access granted by using the access password: kill, rename, write, read, or execute.

random access memory or **RAM**

Semiconductor memory which can be addressed directly and either read from or written to. "User RAM" is that portion of RAM which is left untouched by TRSDOS and DISK BASIC code, from hex 7000 to end of memory.

real-time clock

An interrupt driven routine that keeps time by updating certain memory locations every 25 milliseconds, regardless of what the current background task is. At power-on, the real-time clock is set to 00:00:00. When interrupts are disabled, the clock is stopped.

reset

To press the reset button on the rear left of the TRS-80, next to the Expansion Interface connection. Pressing reset is equivalent to powering up the Computer, except that the contents of user RAM are unaffected.

resident system program

That part of TRSDOS which remains in RAM; the "executive TRSDOS program", which calls in other TRSDOS code as needed.

read-only memory or **ROM**

Pre-programmed semiconductor memory which is directly addressable but can only be read, not written to. The LEVEL II TRS-80 includes 12K of ROM, where a bootstrap program, LEVEL II BASIC, and other code are permanently stored.

routine

A sequence of instructions to carry out a certain function; typically, a routine may be called from multiple points in a program. For example: keyboard scan routine.

sector

One-tenth of a track on a diskette, containing 256 bytes of storage; a TRSDOS "physical record".

sequential access

Reading from a disk file or writing to it "from start to finish", without being able to directly access a particular record in the file.

statement

A complete instruction in BASIC.

string

Any sequence of characters which must be examined verbatim for meaning: in other words, the string does not correspond to a quantity. For example, the *number* 1234 represents the same quantity as 1000+234, but the *string* "1234" does not. (String addition is actually concatenation, or stringing-together, so that: "1234" equals "1" + "2" + "3" + "4").

system file

A TRSDOS disk file with the extension /SYS. Such files are read-protected. To avoid confusion, don't use the extension /SYS on your own disk files.

syntax

The "grammatical" requirements for a command or statement. Syntax generally refers to punctuation and ordering of elements within a statement. See "Notation Conventions", **General Information**, for a description of syntax abbreviations used in this manual.

task

A relatively fundamental routine which the Computer performs periodically or upon request.

track

One of 35 concentric circles on the disk, each of which contains 10 sectors, or 2560 bytes of storage. The tracks are not physical entities like grooves on a record; they are magnetic traces.

transfer address

See **entry point.**

TRSDOS

TRS-80 Disk Operating System, pronounced "triss-doss". TRSDOS is supplied on disk and is then loaded into RAM.

user RAM or **user memory**

See **random access memory.**

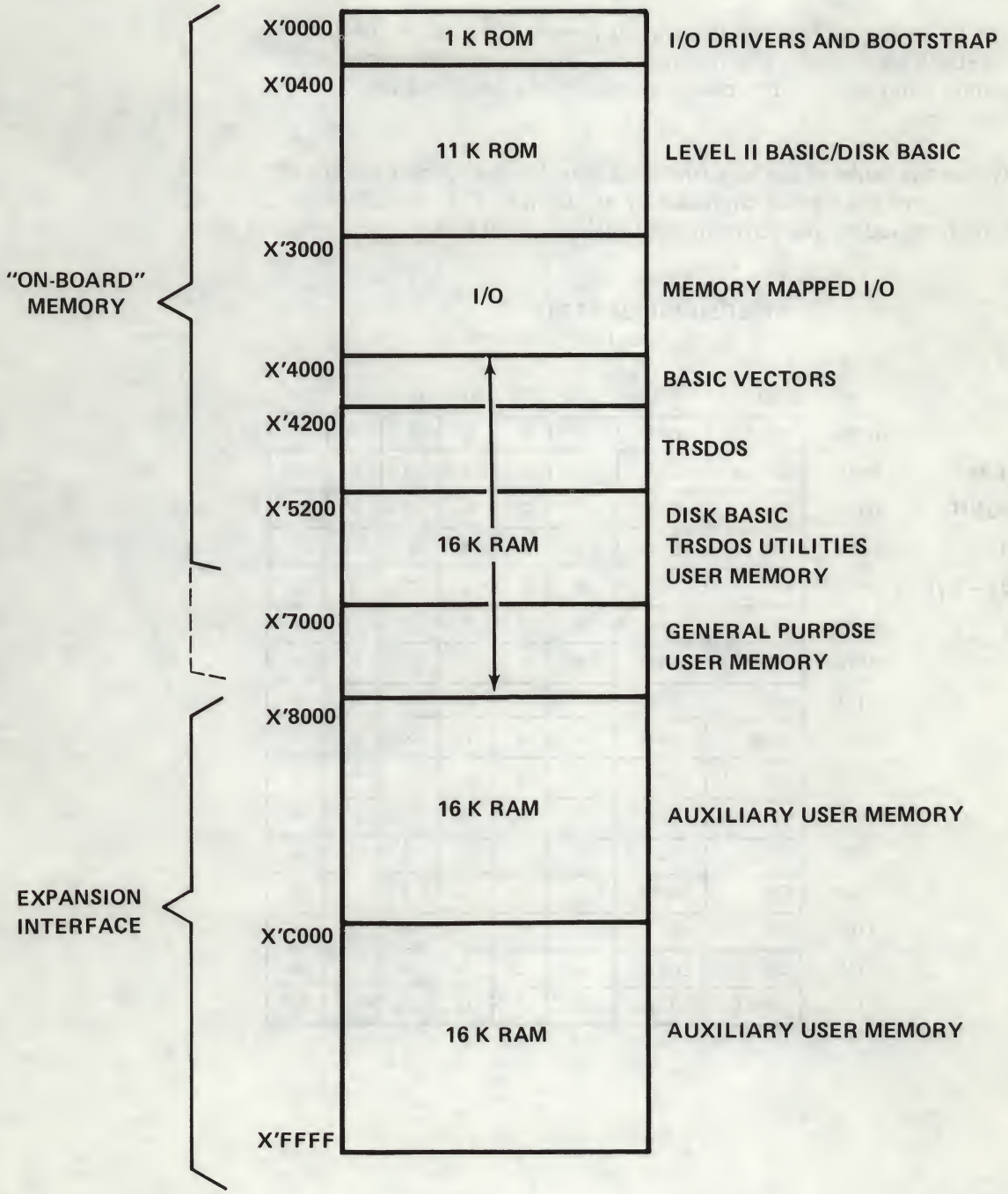
utility

A program or routine which serves a limited, specific purpose. There are two extended TRSDOS utilities, FORMAT and BACKUP, and two non-TRSDOS utilities, DISKDUMP/BAS and TAPEDISK.

write-protect

To physically protect a disk from being written to by placing a tape over the write-protect notch.

Memory Map


X'0000
1 K ROM
I/O DRIVERS AND BOOTSTRAP
X'0400
11 K ROM
LEVEL II BASIC/DISK BASIC
X'3000
I/O
MEMORY MAPPED I/O
"ON-BOARD" MEMORY
X'4000
BASIC VECTORS
X'4200
TRSDOS
X'5200
16 K RAM
DISK BASIC
TRSDOS UTILITIES
USER MEMORY
X'7000
GENERAL PURPOSE
USER MEMORY
X'8000
16 K RAM
AUXILIARY USER MEMORY
EXPANSION INTERFACE
X'C000
16 K RAM
AUXILIARY USER MEMORY
X'FFFF

TRSDOS Character Tables

Bit-Pattern Codes

The following table illustrates the bit pattern for each of the 128 TRSDOS characters. The remaining 128 codes represent special graphics and space compression characters, as described later. See Notes.

To use the table: Combine the most significant and least significant bit-patterns for a given character. For example, the character Q is represented by the pattern: 1010001 (decimal 81).

MOST SIGNIFICANT BITS
($b_7 - b_5$)

LEAST SIGNIF. BITS ($b_4 - b_1$)	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	@	p
0001	BREAK	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BKSP	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	↑	k	↑
1100	FF	HOME	,	<	L	↓	l	↓
1101	CR	BOL	—	=	M	←	m	←
1110	CURON	EREOL	.	>	N	→	n	→
1111	CUROFF	EREOF	/	?	O	_	o	DEL

Decimal/Hexadecimal Codes

Code			Code			Code		
Dec.	Hex.	Char.	Dec.	Hex.	Char.	Dec.	Hex.	Char.
0	00	NULL	32	20	SPACE	64	40	@
1	01	BREAK	33	21	!	65	41	A
2	02	STX	34	22	"	66	42	B
3	03	ETX	35	23	#	67	43	C
4	04	EOT	36	24	$	68	44	D
5	05	ENQ	37	25	%	69	45	E
6	06	ACK	38	26	&	70	46	F
7	07	BEL	39	27	'	71	47	G
8	08	BKSP	40	28	(	72	48	H
9	09	HT	41	29	)	73	49	I
10	0A	LF	42	2A	*	74	4A	J
11	0B	VT	43	2B	+	75	4B	K
12	0C	FF	44	2C	,	76	4C	L
13	0D	CR	45	2D	—	77	4D	M
14	0E	CURON	46	2E	.	78	4E	N
15	0F	CUROFF	47	2F	/	79	4F	O
16	10	DLE	48	30	0	80	50	P
17	11	DC1	49	31	1	81	51	Q
18	12	DC2	50	32	2	82	52	R
19	13	DC3	51	33	3	83	53	S
20	14	DC4	52	34	4	84	54	T
21	15	NAK	53	35	5	85	55	U
22	16	SYN	54	36	6	86	56	V
23	17	ETB	55	37	7	87	57	W
24	18	CAN	56	38	8	88	58	X
25	19	EM	57	39	9	89	59	Y
26	1A	SUB	58	3A	:	90	5A	Z
27	1B	ESC	59	3B	;	91	5B	↕
28	1C	HOME	60	3C	<	92	5C	↕
29	1D	BOL	61	3D	=	93	5D	←
30	1E	EREOL	62	3E	>	94	5E	→
31	1F	EREOF	63	3F	?	95	5F	—

Note: 96-127 (hex 60-7F) are lower-case counterparts to 64-95 (hex 40-5F).; only upper-case characters are displayable.

Notes

The TRSDOS character set may be subdivided into the following functional groups:

decimal code	hex code	function
0-31	00-1F	Control characters
32-95	20-5F	Keyboard/display characters
96-127	60-7F	Non-printing characters (*code-32* is printed)
128-191	80-BF	Graphics characters
192-255	C0-FF	Space-compression codes

The following **control characters** may be entered directly from the keyboard:

character	
BREAK	BREAK
BKSP	←
HT	→
LF	↓
CR	ENTER
CAN	SHIFT ←
EM	SHIFT →
SUB	SHIFT ↓
ESC	SHIFT ↑
EREOF	CLEAR
SP	SPACE-BAR

For a description of the **graphics characters**, run the following program. If you do not have a line printer connected, change all LPRINTs to PRINTs and use the shift-@ key to pause the display.

```
10 CLS: DEFINT A-Z
20 FORI=128 TO 191
30 POKE 15360,I
35 LPRINT CHR$(138)
40 LPRINT"GRAPHICS CODE # ";I
45 LPRINT CHR$(138)
50 A1=POINT(0,0): A2=POINT(1,0)
60 A3=POINT(0,1): A4=POINT(1,1)
70 A5=POINT(0,2): A6=POINT(1,2)
80 LPRINTTAB(8)CHR$(A1*(-40)+48);CHR$(A2*(-40)+48)
90 LPRINTTAB(8)CHR$(A3*(-40)+48);CHR$(A4*(-40)+48)
100 LPRINTTAB(8)CHR$(A5*(-40)+48);CHR$(A6*(-40)+48)
110 NEXT
```

The **space-compression codes** provide a compact means of representing strings of blanks from zero to 63 blanks. For example, C0 represents zero blanks; C1, 1 blank; C2, 2 blanks; FF, 63 blanks.

Base Conversions

The following table lists base conversions for all one-byte values.

DEC.	BINARY	HEX.	OCT.
0	00000000	00	000
1	00000001	01	001
2	00000010	02	002
3	00000011	03	003
4	00000100	04	004
5	00000101	05	005
6	00000110	06	006
7	00000111	07	007
8	00001000	08	010
9	00001001	09	011
10	00001010	0A	012
11	00001011	0B	013
12	00001100	0C	014
13	00001101	0D	015
14	00001110	0E	016
15	00001111	0F	017
16	00010000	10	020
17	00010001	11	021
18	00010010	12	022
19	00010011	13	023
20	00010100	14	024
21	00010101	15	025
22	00010110	16	026
23	00010111	17	027
24	00011000	18	030
25	00011001	19	031
26	00011010	1A	032
27	00011011	1B	033
28	00011100	1C	034
29	00011101	1D	035
30	00011110	1E	036
31	00011111	1F	037
32	00100000	20	040
33	00100001	21	041
34	00100010	22	042
35	00100011	23	043
36	00100100	24	044
37	00100101	25	045
38	00100110	26	046
39	00100111	27	047
40	00101000	28	050
41	00101001	29	051
42	00101010	2A	052
43	00101011	2B	053
44	00101100	2C	054
45	00101101	2D	055
46	00101110	2E	056
47	00101111	2F	057
48	00110000	30	060
49	00110001	31	061
50	00110010	32	062
51	00110011	33	063
52	00110100	34	064
53	00110101	35	065
54	00110110	36	066
55	00110111	37	067
56	00111000	38	070
57	00111001	39	071
58	00111010	3A	072
59	00111011	3B	073
60	00111100	3C	074
61	00111101	3D	075
62	00111110	3E	076
63	00111111	3F	077
64	01000000	40	100
65	01000001	41	101
66	01000010	42	102
67	01000011	43	103
68	01000100	44	104
69	01000101	45	105
70	01000110	46	106
71	01000111	47	107
72	01001000	48	110
73	01001001	49	111
74	01001010	4A	112
75	01001011	4B	113
76	01001100	4C	114
77	01001101	4D	115
78	01001110	4E	116
79	01001111	4F	117
80	01010000	50	120
81	01010001	51	121
82	01010010	52	122
83	01010011	53	123
84	01010100	54	124
85	01010101	55	125

DEC.	BINARY	HEX.	OCT.
86	01010110	56	126
87	01010111	57	127
88	01011000	58	130
89	01011001	59	131
90	01011010	5A	132
91	01011011	5B	133
92	01011100	5C	134
93	01011101	5D	135
94	01011110	5E	136
95	01011111	5F	137
96	01100000	60	140
97	01100001	61	141
98	01100010	62	142
99	01100011	63	143
100	01100100	64	144
101	01100101	65	145
102	01100110	66	146
103	01100111	67	147
104	01101000	68	150
105	01101001	69	151
106	01101010	6A	152
107	01101011	6B	153
108	01101100	6C	154
109	01101101	6D	155
110	01101110	6E	156
111	01101111	6F	157
112	01110000	70	160
113	01110001	71	161
114	01110010	72	162
115	01110011	73	163
116	01110100	74	164
117	01110101	75	165
118	01110110	76	166
119	01110111	77	167
120	01111000	78	170
121	01111001	79	171
122	01111010	7A	172
123	01111011	7B	173
124	01111100	7C	174
125	01111101	7D	175
126	01111110	7E	176
127	01111111	7F	177
128	10000000	80	200
129	10000001	81	201
130	10000010	82	202
131	10000011	83	203
132	10000100	84	204
133	10000101	85	205

DEC.	BINARY	HEX.	OCT.
134	10000110	86	206
135	10000111	87	207
136	10001000	88	210
137	10001001	89	211
138	10001010	8A	212
139	10001011	8B	213
140	10001100	8C	214
141	10001101	8D	215
142	10001110	8E	216
143	10001111	8F	217
144	10010000	90	220
145	10010001	91	221
146	10010010	92	222
147	10010011	93	223
148	10010100	94	224
149	10010101	95	225
150	10010110	96	226
151	10010111	97	227
152	10011000	98	230
153	10011001	99	231
154	10011010	9A	232
155	10011011	9B	233
156	10011100	9C	234
157	10011101	9D	235
158	10011110	9E	236
159	10011111	9F	237
160	10100000	A0	240
161	10100001	A1	241
162	10100010	A2	242
163	10100011	A3	243
164	10100100	A4	244
165	10100101	A5	245
166	10100110	A6	246
167	10100111	A7	247
168	10101000	A8	250
169	10101001	A9	251
170	10101010	AA	252
171	10101011	AB	253
172	10101100	AC	254
173	10101101	AD	255
174	10101110	AE	256
175	10101111	AF	257
176	10110000	B0	260
177	10110001	B1	261
178	10110010	B2	262
179	10110011	B3	263
180	10110100	B4	264
181	10110101	B5	265
182	10110110	B6	266

DEC.	BINARY	HEX.	OCT.
183	10110111	B7	267
184	10111000	B8	270
185	10111001	B9	271
186	10111010	BA	272
187	10111011	BB	273
188	10111100	BC	274
189	10111101	BD	275
190	10111110	BE	276
191	10111111	BF	277
192	11000000	C0	300
193	11000001	C1	301
194	11000010	C2	302
195	11000011	C3	303
196	11000100	C4	304
197	11000101	C5	305
198	11000110	C6	306
199	11000111	C7	307
200	11001000	C8	310
201	11001001	C9	311
202	11001010	CA	312
203	11001011	CB	313
204	11001100	CC	314
205	11001101	CD	315
206	11001110	CE	316
207	11001111	CF	317
208	11010000	D0	320
209	11010001	D1	321
210	11010010	D2	322
211	11010011	D3	323
212	11010100	D4	324
213	11010101	D5	325
214	11010110	D6	326
215	11010111	D7	327
216	11011000	D8	330
217	11011001	D9	331
218	11011010	DA	332

DEC.	BINARY	HEX.	OCT.
219	11011011	DB	333
220	11011100	DC	334
221	11011101	DD	335
222	11011110	DE	336
223	11011111	DF	337
224	11100000	E0	340
225	11100001	E1	341
226	11100010	E2	342
227	11100011	E3	343
228	11100100	E4	344
229	11100101	E5	345
230	11100110	E6	346
231	11100111	E7	347
232	11101000	E8	350
233	11101001	E9	351
234	11101010	EA	352
235	11101011	EB	353
236	11101100	EC	354
237	11101101	ED	355
238	11101110	EE	356
239	11101111	EF	357
240	11110000	F0	360
241	11110001	F1	361
242	11110010	F2	362
243	11110011	F3	363
244	11110100	F4	364
245	11110101	F5	365
246	11110110	F6	366
247	11110111	F7	367
248	11111000	F8	370
249	11111001	F9	371
250	11111010	FA	372
251	11111011	FB	373
252	11111100	FC	374
253	11111101	FD	375
254	11111110	FE	376
255	11111111	FF	377

Index for TRSDOS & DISK BASIC Reference Manual



INDEX

For: TRSDOS Version 2.1
DISK BASIC Version 1.1

Index

Subject	Page
&H, BASIC hex constant prefix	7-6
&O, BASIC octal constant prefix	7-6
<EN>, carriage return character	7-38
<LF>, line-feed character	7-38
access,	8-2
random	7-65, 8-11
sequential	7-60, 8-10
address	8-2
alphanumeric	8-2
argument	8-2
array	8-2
-notation	1-4
ASCII	8-2
format	7-32, 8-2
assembly-language	
I/O, TRSDOS	6-5
access from BASIC	7-14, 7-20
ATTRIB, TRSDOS command	4-12
AUTO, TRSDOS command	4-11
BACKUP, TRSDOS utility	
abbreviated instructions for	2-16
detailed description of	5-2
Important Notice	2-17
background task	8-2
backup disk	8-3
Base Conversions,	
decimal/binary/octal/hex	8-18
BASIC, TRSDOS	
command file	1-2, 3-4, 7-2
BASIC2, TRSDOS command	4-2
baud	8-3
binary	8-3
bit	8-3
blocking of logical records	
under TRSDOS	6-3, 6-7
bootstrap program	8-3
break	8-3
buffer	6-5, 6-6, 7-2, 7-33 ff, 8-3
byte	8-4
cassette I/O under	
DISK BASIC	7-5, 7-10

Subject	Page
CLOCK, TRSDOS command	4-14
clock, real time, see real-time clock	
CLOSE, BASIC statement	7-36
close a file	6-11, 7-36, 8-4
CMD"D", BASIC statement	7-7
CMD"R", BASIC statement	7-10
CMD"S", BASIC statement	7-10
CMD"T", BASIC statement	7-10
command	
file	3-7, 8-4
format	3-5
command,	
TRSDOS library	4-2 ff, 8-8
TRSDOS system	4-11 ff
compressed format, BASIC	7-31, 8-4
COPY, TRSDOS command	4-15
CVD, BASIC function	7-54
CVI, BASIC function	7-54
CVS, BASIC function	7-54
data	8-4
data/device control block	
(DCB)	6-6, 8-4
data diskette	5-4
DATE, TRSDOS command	4-15
DEBUG, TRSDOS command	4-3
debug	8-4
decimal	8-5
default	8-5
delimiter,	6-6, 8-5
BASIC INPUT #	7-40
DEFFN, BASIC statement	7-11
DEFUSR, BASIC statement	7-14
destination	8-5
diskette	5-2
DEVICE, TRSDOS command	4-16
device	8-5
DIR, TRSDOS command,	4-16
directory	8-5

Subject	Page
DISK BASIC	
error messages	.7-77
ROM calls	.7-22
versions and releases	1-6
DISKDUMP/BAS, auxiliary utility program	.5-8 ff
diskette,	
data	.2-10
TRSDOS software	.2-10
diskette	.2-5 ff, 8-5
care	2-8
organization	2-6, 6-2
specifications	.2-10
drive number	8-6
drive numbering	2-5
drive specification	3-6, 3-8, 8-5
drive zero	2-5
dummy variable	8-6
DUMP, TRSDOS command	.4-18
end of file (EOF)	.4-16, 6-6, 6-9, 7-55, 8-6
entry point	.4-18, 5-6, 6-8, 7-14, 8-6
EOF, BASIC function	.7-55
ERROR, BASIC statement	7-6
error messages,	
BASIC	.7-6, 7-77
TRSDOS	.6-12
expression	8-6
FIELD, BASIC statement	.7-47
field,	8-6
overlapping	.7-76
field name	.7-48, 7-67, 8-6
file, TRSDOS	.3-3, 6-3 ff, 8-6
extension	.3-6 ff, 7-32, 8-6
name	3-6, 8-7
specification	.3-6 ff, 6-6, 8-7
foreground task	.3-2, 4-9, 4-10, 8-7
FORMAT, TRSDOS utility	.5-4 ff
format	.5-4 ff, 8-7
FREE, TRSDOS command	.4-19

Subject	Page
GET, BASIC statement	.7-49
granule	.4-16, 4-19, 5-4, 6-3, 6-12, 8-7
hexadecimal	8-7
constants, BASIC	7-6
increment	8-7
input	8-7
INPUT #, BASIC statement	.7-37
INSTR, BASIC function	.7-15
interrupt	.7-5, 7-10, 8-8
KILL, BASIC command	.7-28
KILL, TRSDOS command	.4-19
kilobyte	8-8
LIB, TRSDOS command	.4-19
library command, TRSDOS	.4-11
LINE INPUT, BASIC statement	.7-16
LINE INPUT #, BASIC statement	.7-42
LIST, TRSDOS command	.4-20
LOAD, BASIC command	.7-28
LOAD, TRSDOS command	.4-20
LOF, BASIC function	.7-56
LSET, BASIC statement	.7-53
machine language,	8-8
dump to disk	.4-18
load from disk	.4-20
reserve RAM for	7-3
access routines from BASIC	.7-14, 7-20
Master Password	.4-21, 5-2
MERGE, BASIC command	.7-29
MID$=, BASIC statement	.7-17
Mini Disk	.2-1 ff
connection	2-3
operation	2-5
specifications	.2-10
MKD$, BASIC function	.7-57
MKI$, BASIC function	.7-57
MKS$, BASIC function	.7-57
Notation	1-3, 6-5
null string	8-8

Index

Subject	Page
object code	8-8 and see machine language
octal	8-9
constants, BASIC	7-6
OPEN, BASIC statement	.7-34
open a file	.6-6, 6-9, 7-33, 7-34, 8-9
parameter	8-9
password	.3-6, 3-8, 4-12, 4-13, 8-9
PRINT, TRSDOS command	.4-21
PRINT #, BASIC statement	.7-43
prompt	.8-10
PROT, TRSDOS command	.4-21
protected file	.6-12, 8-10
protection level	.4-12, 4-22, 8-10
PUT, BASIC statement	.7-50
random access	.7-65
random access memory (RAM)	.8-10
allocation	1-2, 3-4, 6-2
random access memory (RAM), BASIC program storage in	.7-32
reserving for machine language code under DISK BASIC	7-3
read only memory (ROM)	.8-10
calls from BASIC	.7-22
calls from TRSDOS	.6-8 ff
organization	6-2
real-time clock, in Expansion Interface	1-2
memory location of	4-9
to display	.4-14
to set	.4-23
to turn off	.7-10
to turn on	.7-10
record, TRSDOS physical	6-4, 6-7, 8-9
logical	6-7, 8-8
release	1-6
RENAME, TRSDOS command	.4-22
reset	.8-10
resident program	.3-4, 8-10
ROM (see read only memory)	
RSET, BASIC statement	.7-53
RUN"program", BASIC command	.7-31
SAVE, BASIC command	.7-31
sector, diskette	.2-6, 8-11
sequential access	.7-60, 8-11
statement, BASIC	.8-11
string	.8-11
subrecord	.7-71
syntax,	.8-11
TRSDOS command	3-5
system command	.4-22
file	.3-7, 8-11
routine	.6-5 ff
TAPEDISK, auxiliary utility	5-6
task, background	8-2
foreground	.3-2, 4-9, 4-10, 4-14, 8-7
TIME, TRSDOS command	.4-23
TIME$, BASIC function	.7-19
TRACE, TRSDOS command	.4-10
transfer address	.4-18, 8-11
track, diskette	.2-6, 8-11
TRSDOS assembly I/O	.6-5 ff
error messages	.6-12
file specification	3-6
library commands	.4-11
memory organization	6-2
RAM allocation	.3-4, 8-13
system commands	4-2
utilities	.5-1 ff
versions and releases	1-6
USING, BASIC PRINT format modifier	.7-46
USR, BASIC function	.7-20
utility TRSDOS	5-2
auxiliary	5-6
VERIFY, TRSDOS command	.4-24
versions and releases	1-6
write protect	2-6

IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

LIMITED WARRANTY

Radio Shack warrants for a period of 90 days from the date of delivery to customer that the computer hardware described herein shall be free from defects in material and workmanship under normal use and service. This warranty shall be void if this unit's case or cabinet is opened or if the unit is altered or modified. During this period, if a defect should occur, the product must be returned to a Radio Shack store or dealer for repair. Customer's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at Radio Shack's election and sole expense, except there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including but not limited to statements regarding capacity, suitability for use, or performance of the equipment, shall be or be deemed to be a warranty or representation by Radio Shack, for any purpose, nor give rise to any liability or obligation of Radio Shack whatsoever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

RADIO SHACK TC **A DIVISION OF TANDY CORPORATION**

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.

Name: _____

Company: _____

Mailing Address: _____

_____

Phone: _____

Brief list of hardware in your system:

_____

_____

_____

_____

Please help us better serve your needs – answer these survey questions and return the survey to us as as soon as possible.

1. What do/will you use your TRS-80 TRSDOS System for? _____

_____

2. What is your approximate investment in hardware $_____ and software $ _____ ?

3. What area(s) of your TRS-80 system needs the most improvement? _____

_____

_____

_____

_____

_____

4. What approach would be the most effective for accomplishing those improvements? _____

_____

_____

_____

_____

_____

5. What would be the result if these improvements were made? _____

_____

_____

_____

Thank you.

(Fold Along This Line)

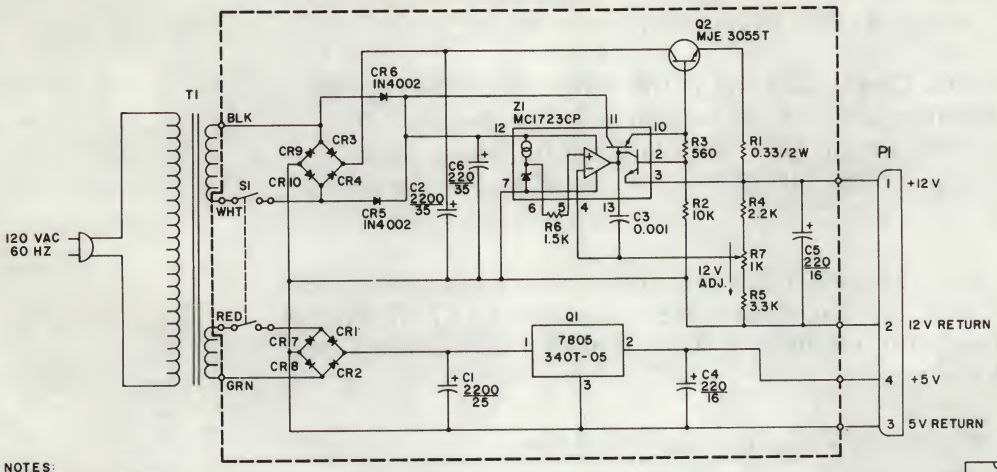
Place
Stamp
Here

3/25/85
25
75

TRS-80 Computer Survey
1100 One Tandy Center
Fort Worth, TX 76102

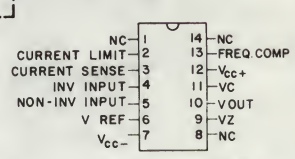
(Fold Along This Line)

Power Supply



NOTES:
UNLESS OTHERWISE SPECIFIED:

1. ALL RESISTORS ARE 1/4 WATT, 5%, RESISTOR VALUES IN OHMS, K=1000.
2. CAPACITOR VALUES IN MICROFARADS AND WORKING VOLTAGE.
3. ⟶ INDICATES CLOCKWISE ROTATION.



MC1723CP TOP VIEW

Making a TRSDOS BACKUP

Before you do anything else with your TRSDOS diskette, follow these instructions for making a "safe copy" of your system software. That way, if anything should happen to your original, you won't be "out of business" while you wait to get another one.

Connect the Mini Disk system and power it up as described in the Mini Disk Operation chapter. Be sure your TRSDOS diskette is in Drive 0 when you turn on the CPU. (And just for safety, leave the write protect tape on the TRSDOS diskette until you've duplicated it.)

If you have more than one drives connected, place a blank diskette in drive 1. If not, have the blank diskette handy – BACKUP will tell you when to insert it into drive 0. Do not place a write protect tape on the blank diskette.

After you power on the CPU, the display will read

```
TRSDOS - DISK OPERATING SYSTEM - VER 2.1

DOS READY
-
```

Type:

```
BACKUP ENTER
```

The system will then display:

```
TRSDOS DISK BACKUP UTILITY  VER 2.1
```

If you have only 1 drive connected, type:

```
SOURCE DRIVE NUMBER ? 0 ENTER
DESTINATION DRIVE NUMBER ? 0 ENTER
```

If you have two or more drives, type:

```
SOURCE DRIVE NUMBER ? 0 ENTER
DESTINATION DRIVE NUMBER ? 1 ENTER
```

Now type in the date in MM/DD/YY form. For example, if it's August 3, 1978, type:

```
BACKUP DATE (MM/DD/YY) ? 08/03/78 ENTER
```

TRSDOS will then start the BACKUP procedure. First it will format the blank diskette, locking out any defective tracks; then it will duplicate the contents of the TRSDOS diskette onto it.

If you are using only one drive, BACKUP will tell you when to insert the destination (blank) diskette, and when to re-insert the source (TRSDOS) diskette. During the BACKUP process, you will have to swap the two diskettes several times.

When the process is completed, the message:

```
BACKUP COMPLETE - PRESS ENTER TO CONTINUE
```

will be displayed.

If TRSDOS instead displays the message:

```
BACKUP REJECTED DUE TO (...)
```

then erase the diskette with a bulk eraser (Radio Shack Catalog Number 44-210) and repeat the BACKUP procedure. If it still won't work, you may need to try using another blank diskette.

IMPORTANT NOTICE

The BACKUP utility is provided solely for your personal use in maintaining safe copies of your TRSDOS and data diskettes. BACKUP automatically places copyrighted TRSDOS software onto each destination disk. TRSDOS users may BACKUP the system software solely for personal use.

See the Copyright Notice at the beginning of this Manual for more details.

TRSDOS
An Overview



TRSDOS

Contents of This Section

Introduction 2
Entering a Command 5
File Specification 6

Introduction

TRSDOS, like the entire TRS-80 Microcomputer System, is designed to satisfy a broad range of users, including:

- The novice to computers, who wants to start simply and learn the details gradually
- The experienced programmer, who expects to write complex programs, and may want to use some of the system routines on a machine language level, to accomplish a variety of sophisticated, customized applications
- The pure "user", who is only interested in using programs, not writing them (for example, a clerk using an inventory program on the office TRS-80).

What Is an Operating System?

By the time you finish this book, you'll have a pretty good idea . . . But for the time being, here's an overview.

An operating system is a master program that allows a complex computer system, including various Input/Output (I/O) devices, storage devices and programs, to interact efficiently and with apparent simplicity. The operating system makes sure everything that has to be done, gets done – and you don't even have to know what it is that "has to get done"!

Here's a rather arbitrary breakdown of what an operating system does (see **Glossary** for unfamiliar terms):

- Interfaces the central processing unit (CPU) with the various input/output and storage devices
- Accepts and interprets operator commands
- "Shepherds" your programs (and system utilities you request) in and out of the execution sequence, by allocating CPU time, I/O channels, storage and other system resources
- Handles interrupts, and oversees the execution of both foreground and background tasks
- Provides fundamental routines which would otherwise have to be included in every program; this saves memory and programming time

You don't always have to be aware of the operating system to use it. For example, when you're using DISK BASIC, you don't see TRSDOS at all. But the system is still there, executing a program called BASIC; BASIC, in turn, executes your own programs and commands.

At other times, the operating system may be quite visible to you, allowing you to enter system commands directly. This is the case with TRSDOS and its "DOS READY" mode.

What Is TRSDOS?

The TRS-80 Disk Operating System (TRSDOS) is a comprehensive set of system routines and file management utilities. Much of its complexity (and power) relates to the fact that it is disk-based.

The system is loaded from diskette, and uses diskettes to store internal bookkeeping information as well as data and programs you create. TRSDOS uses completely dynamic disk space allocation, so you can open and manipulate files freely without worrying where they are physically located on the diskette. When a file fills the space currently allocated to it, TRSDOS automatically finds and acquires more space to accommodate additional data (assuming space is available on the diskette).

(All information on a diskette – programs, data, and TRSDOS itself – exists in the form of files. For more information on files, see the **Glossary**, **Files Entry**, and the **Technical Information** chapter.)

In addition to system routines which perform the functions described above under "What is an Operating System?", TRSDOS includes several file management utilities to let you manipulate and modify existing files on the diskette: copy, append, rename, change the protection status, etc.

How TRSDOS uses RAM

TRSDOS consists of:

- an executive program file
- auxiliary system-routine files
- a library-command file
- extended utility files (BACKUP and FORMAT)
- and the DISK BASIC file.

The **executive program** is loaded into RAM on power-up, and remains there at all times while TRSDOS is running. For this reason it is called the "**resident**" TRSDOS program. It includes certain system routines, tables, pointers, and Input/Output drivers.

The **auxiliary system files** contain routines and commands which are loaded as needed to execute your commands and programs. These routines load into an "overlay" area of memory. When TRSDOS has executed the routine, another one may be loaded in the same area, or "overlayed". The use of overlays means that execution of system routines will not affect your memory area (addresses above 51FF hex).

The **library command file** contains the routines for executing most of the operator commands. These routines load into memory addresses from 5200 to 6FFF. Therefore your machine language programs should generally be located above 6FFF. That way they won't be affected by execution of the library commands.

The TRSDOS **extended utility programs** are loaded when you type in their file names, BACKUP and FORMAT. These programs can use all available memory – even the resident TRSDOS program is wiped out when they are loaded.

DISK BASIC is a set of enhancements to LEVEL II BASIC. When you type in its file name, BASIC, it will load into memory beginning at 5200, and begin execution.

Entering a Command

Whenever the prompt,

```
DOS READY
_
```

is displayed, you may enter an operator command. In its simplest form, an operator command is just a single word – a system or library command, the name of an extended utility program, or the name of a user command program. All these categories will be detailed later.

As an example,

```
DIR ENTER
```

tells TRSDOS to display the user file directory for drive 0.

In general, operator commands will require more than one word; for example, to kill (delete) a certain file, you have to specify the file name.

```
KILL XYZ ENTER
```

tells TRSDOS to find the file named XYZ, eliminate it from the directory of the diskette which contains it, and release the space occupied by that file.

In general, an operator command consists of a command followed by one or more file specifications, followed by special parameters:

command [b*filespec*] [b(*param*)] [bTO] [b*filespec*] [b(*param*)]

where *filespec* is a valid TRSDOS file specification (more below)
param is a parameter which details how the command affects the specified file(s).

If this command format seems complex, don't worry; that's because it's so generalized. The actual commands can be quite simple, as you'll see from the examples given with each command.

Whenever you finish typing in a command, press ENTER .
TRSDOS will then process the command as follows:

1) Check to see if it's a system or library command; if so, execute it immediately . . . otherwise
2) Check to see if it's the name of a utility program; if so, execute it via the extended utility package . . . otherwise
3) Examine the diskette directory on each drive to see if the command is listed as a user command file; if so, load and execute the file.

File Specification

A file specification (filespec) is the way you reference a particular file, whether you're operating under TRSDOS, DISK BASIC, or any other command program (e.g., TAPEDISK).

Disk file specifications have the following format:

name[/*ext*] [.*pw*] [:*d*]

where

name is the file name, consisting of from 1 to 8 alphanumeric characters, the first of which must be alphabetic

ext is an optional extension of the name, consisting of from 1 to 3 alphanumeric characters, the first of which must be alphabetic. The extension, if used, must be preceded by a slash symbol.

pw is an optional password, consisting of from 1 to 8 alphanumeric characters, the first of which must be alphabetic. The password, if used, must be preceded by a period symbol.

:*d* is an optional drive specification, with *d* equal to 0,1,2 or 3, depending on which drive you wish to specify. The drive specification, if used, must be preceded by a colon.

Do not embed blanks in a file specification. If you do, TRSDOS will terminate the filespec at the first blank; if the truncated filespec is valid, you won't receive an error message.

Valid file names:

A	INVNTORY	DATA11
GAMES/BAS	SORTER/VR1	SORTER/VR2
PAYROLL/BAS.SESAME	SECRETS.MYNAME	POETRY/TXT:1
DRIVECHK:1	DRIVECHK:2	AUG3078/DAT.JQD
AUG1578	TAXES/TXT.TEAPARTY:1	CHKWRITR/BAS.VERSION2

To take a completely "filled out" filespec, TAXES/TXT.TEAPARTY:1 refers to a file named TAXES, with an extender TXT, and a password TEAPARTY. This file is referenced to drive 1. If you are creating a file under that filespec, it will be placed on drive 1. If you are reading or writing to the file specified, TRSDOS will reference drive 1 for the file.

What makes a particular filespec unique?

The name, extension and drivespec all figure into the uniqueness of a particular filespec. The password does not.

For example, the following filespecs refer to distinct files:

A	A/BAS	A/CMD	
DRIVECHK:0	DRIVECHK:1	DRIVECHK:2	DRIVECHK:3

However, the following filespecs *cannot* be used to reference distinct files:

RECEIPTS RECEIPTS.AUG3078 RECEIPTS.AUG3178

(There *are* cases where two different passwords are used to access the same file; see **TRSDOS Library Commands,** ATTRIB.)

More on Extensions

The particular extension you use can be purely arbitrary and personalized. Used this way, extensions give you an extra three characters to work with in creating a suitable file name.

Examples:

PAYROLL/AUG PAYROLL/SEP PAYROLL/OCT

However, extensions become more meaningful when they are used as type specifiers, using some convention. Here's a recommended set of extensions:

/BAS	BASIC program file stored in compressed format
/TXT	ASCII text: BASIC program saved in ASCII form, or source file, etc.
/CMD	machine language command file
/CIM	core (RAM) image file, not necessarily executable
/REL	relocatable machine language program file
/SYS	system program – files which are part of TRSDOS. Don't use for your files.
/OV*n*	overlay number *n*
/DVR	I/O driver module

One advantage of this usage is that anyone looking at a directory listing of a diskette will know what kinds of programs he's looking at.

Another advantage is that TRSDOS is equipped to recognize certain extensions. For example, if a file has the extension /CMD, then TRSDOS will load and attempt to execute that file when you type:

filename **ENTER**

omitting the extension /CMD.
That's why you can execute the file BASIC/CMD by typing

BASIC **ENTER**

Similarly, your own programs can be written to recognize extensions.

More on Drive Specifications

If you give a drive specification, TRSDOS will use the specified drive in executing the command. If you omit a drivespec, TRSDOS will search through the directories of all drives in use, starting with drive 0; the first drive with the correct name/extension will be used. However, if the command requires a file creation, TRSDOS will skip over to the first non write-protected diskette.

For example, suppose four files named DRIVECHK are contained on drives 0 through 3. Then every reference to DRIVECHK (no drivespec) would go to drive 0. The filespecs DRIVECHK:0, DRIVECHK:1, DRIVECHK:2, DRIVECHK:3, would allow each of the four files to be accessed.

More on Passwords

The password is assigned when the file is created, and may be changed via the ATTRIB or PROT commands. Files with passwords can only be accessed by reference to the password, or to the diskette's Master Password. So if you assign a password to a file, don't forget it!

It's important to realize that every file has a password, even if you do not specify it explicitly when the file is created. In such cases, a field of 8 blanks becomes the password.

For example, if SAMPLE (a file with no explicit password) exists and you attempt to create a new file, SAMPLE.WATERBOY, TRSDOS will give you a FILE ACCESS DENIED message, since in effect you're trying to access an existing file with the wrong password. The correct password is a string of 8 blanks – which you can omit from the file specification, since 8-blanks is the default password.

TRSDOS Commands



TRSDOS

Contents of This Section

System Commands 2

BASIC2 2
DEBUG 3
TRACE 10

Library Commands 11

AUTO 11
ATTRIB 12
CLOCK 14
COPY 15
DATE 15
DEVICE 16
DIR 16
DUMP 18
KILL 19
FREE 19
LIB 19
LIST 20
LOAD 20
PRINT 21
PROT 21
RENAME 22
TIME 23
VERIFY 24

System Commands

These three commands (BASIC2, DEBUG, TRACE) leave user RAM (hex address 5200-End) "untouched". The necessary code for these commands loads into the overlay area between the resident program and hex 5200. The other commands, referred to as **library commands**, use addresses between hex 5200-6FFF. So locate your machine-language routines above hex 7000 to protect them from the utility commands.

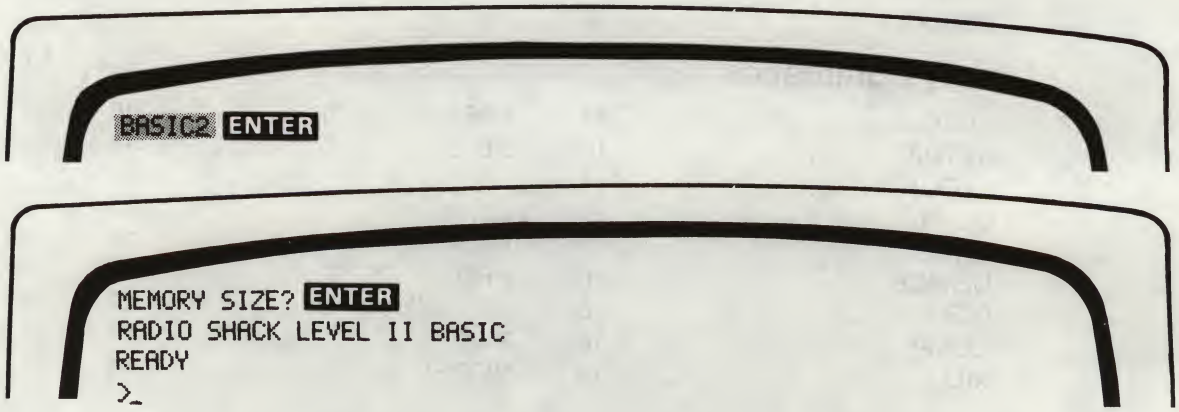
BASIC2 (jump to LEVEL II BASIC)

BASIC2

This command has no arguments or parameters. It simply transfers control to LEVEL II BASIC. Once it has been executed, TRSDOS is no longer resident in RAM. Your TRS-80 will then function as a LEVEL II machine.

You may want to do this to gain memory for programs which don't require disk capabilities. Another possible application would be to LOAD a machine language routine from disk into high memory, and then jump to LEVEL II BASIC via BASIC2, so you can access the routine from LEVEL II, via a USR function.

Example:



To re-load TRSDOS, press the Reset button or type

SYSTEM ENTER

*? /0 ENTER

DEBUG (real-time debugging program)

DEBUG[ϕ(*param*)]

where *param* = ON or OFF, and ON is the default.

DEBUG is a real-time debugging package for use with machine language programs, including both foreground tasks and background programs. (See **Glossary**.) DEBUG lets you examine and alter the contents of the Z-80 registers and RAM locations; jump to specified addresses and begin execution with optional breakpoints; step through programs one instruction (or one CALL) at a time, and more.

All address and byte values in this DEBUG section are given in hexadecimal form – which is the form required by DEBUG.

DEBUG loads into the overlay area; addresses above 51FF are unaffected.

Type:

DEBUG ENTER

to enable the debugging facility. Normal TRSDOS command interpretation continues; but the debug program is now set to load and execute under any of the following conditions:

1. When the BREAK key is pressed.
2. After a program is loaded and before its first instruction is executed.
3. Upon detection of a disk-related error.

Note: TRSDOS system routines and execute-only user routines cannot be fully debugged: you can use DEBUG to examine/alter register and RAM contents, but not to single-step, jump, etc., when these protected programs are the "targets" for DEBUG. Furthermore, since DEBUG loads into the overlay area of RAM, you can't use it with other overlay programs and routines.

DEBUG offers two display formats:

register display with indirect RAM
plus any 64-byte "page" of RAM;
full screen, 256-byte page of RAM.

In the register display format, DEBUG displays all the Z-80 registers, organized for interpretation either as two 8-bit registers or as 16-bit register pairs. Since most programs use several sets of register pairs as indirect pointers or indexing registers, 16 bytes of indirect data are presented with each register pair. Each of the flag registers is shown with an ASCII representation of its flag bits.

An additional 64 bytes of memory are displayed in four lines at the bottom of the display.

Here's a typical DEBUG display sequence. Note that the values in your display will typically vary from these.

```
DEBUG ENTER
DOS READY
BREAK
```

```
AF = 3E 28 --1-1---
BC = 0A 3E =>  09 BA C2 60 09 7D 93 C2  60 09 C9 21 27 41 CD D3
DE = 01 04 =>  1A 4D 45 4D 4F 52 59 20  53 49 5A 45 00 52 41 44
HL = 00 54 =>  01 01 5B 1B 0A 1A 08 18  09 19 20 20 0B 78 B1 20
AF'= FF FF SZ1H1PNC
BC'= 4D BE =>  51 51 CD FC 51 7E 23 18  EC 02 02 00 4E 03 32 E7
DE'= 01 07 =>  4D 4F 52 59 20 53 49 5A  45 00 52 41 44 49 4F 20
HL'= 4D 00 =>  F2 51 06 10 CD 65 51 3A  5D 40 FE 41 20 13 CD F2
IX = 40 15 =>  01 E3 03 00 00 00 4B 49  07 58 04 31 3E 20 44 4F
IY = FF FF =>  FF |F3| AF C3 74 06 C3 00  40 C3 00 40 E1 E9 C3 9F
SP = 41 E8 =>  52 04 DD 03 15 40 15 40  18 43 3F 3F 4C 00 E3 05
PC = 00 60 =>  0B 78 B1 20 FB C9 31 00  06 3A EC 37 3C FE 02 D2
     1010 =>  28 10 FE 44 28 0C FE 30  28 F0 FE 2C 28 EC FE 2E
     1020 =>  20 03 2B 36 30 7B E6 10  28 03 2B 36 24 7B E6 04
     1030 =>  C0 2B 70 C9 32 D8 40 21  30 41 36 20 C9 FE 05 E5
     1040 =>  DE 00 17 57 14 CD 01 12  01 00 03 82 FA 57 10 14_
```

In this display, register B contains the hex value 0A, and register C contains 3E. Taking the BC register pair as a pointer, it points to address 0A3E. Therefore, the contents of memory locations 0A3E through 0A4D are shown to the right of the BC = 0A3E =>marker. In this case, address 0A3E contains 09, 0A3F contains BA, etc.

The flag registers F and F′ are handled differently. For these registers, the hex contents of the flag register is displayed, along with a bit-by-bit alphabetic code which makes it easier to interpret the flag status. For example, bit 7 (leftmost bit) is the sign bit, so the alphabetic code shows an S in that position whenever this bit is "set". Here's a complete table of codes for all the flag bits:

bit status	if set	if not set
7 Sign	S	–
6 Zero	Z	–
5 unused	1	–
4 Half-carry	H	–
3 unused	1	–
2 Parity/overflow	P	–
1 Negative	N	–
0 Carry	C	–

In the above display, none of the F flag bits are set (discounting the unused bits 5 and 3), and all of the F′ flag bits are set.

Notice the four additional lines below the PC register display. Each line shows the contents of 16 bytes, starting at the address to the left of the arrow; the four lines always show a total of 64 bytes of contiguous memory i.e., locations with sequential addresses. The starting point in this four-line display is either 0000 or the last command you specified with the D command (more later.)

The blank area in the lower left of the Display is where commands you enter will be displayed.

DEBUG Commands

Note that some commands are executed as soon as you press the specified command key; other commands are executed only when you hit <SPACE> or ENTER , as indicated below.

Command	Entry Required	Operation Performed
A	none	Shows the ASCII or graphics character corresponding to each value displayed. Shows a period when the value is not displayable as an ASCII or graphics character.
C	none	Single-steps next instruction, with CALLS executed in full. (Next instruction is defined by PC register.) Target program cannot be a system or execute-only file.
D*aaaa*	<SPACE>	Sets memory display starting address to *aaaa*. In full screen mode, sets starting address so *aaaa* is contained in display.
G*aaaa*[,*bbbb*[,*cccc*]]	ENTER	Place *aaaa* in PC register and executes with optional breakpoints at *bbbb* and *cccc*.
H	none	Displays all memory and register values in hexadecimal form.
I	none	Single-steps next instruction (defined by PC register). Target program must not be read-protected.
M[*aaaa*]	<SPACE>	Sets the current modification address to *aaaa*. The modification dialog will then be displayed in the lower left of the screen. If *aaaa* is omitted, the last modification address will be used for *aaaa*. If *aaaa* is currently in the display, its contents will be surrounded by a pair of vertical bars.

Command	Entry Required	Operation Performed
R*rp*b*dddd*	<SPACE>	Loads register pair *rp* with the value *dddd.* *rp* may be any register pair: AF, BC, AF′, BC′, IX, IY, PC, etc.
S	none	Sets display to full screen memory mode, showing 256 contiguous bytes. Press X to return to register display format.
U	none	Dynamic display update mode: lets you observe the execution of a foreground task. Hold down any key for a couple of seconds to exit this mode.
X	none	Sets display to register format; also cancels any command you are in the process of entering, except R-command.
;	none	Increments memory display by one page (in register display mode, page = 64 bytes; page = 256 bytes in full screen mode).
–	none	Decrements memory addresses displayed by one page.

Note: You cannot use the backspace key (←) to delete mistakes made while entering commands. Instead, just hit the X key to cancel the command. Or, if you made the error while typing an address or value, just type the correct address immediately after the incorrect address. DEBUG will only look at the last four digits entered.

For example,

```
D474080  <SPACE>
```

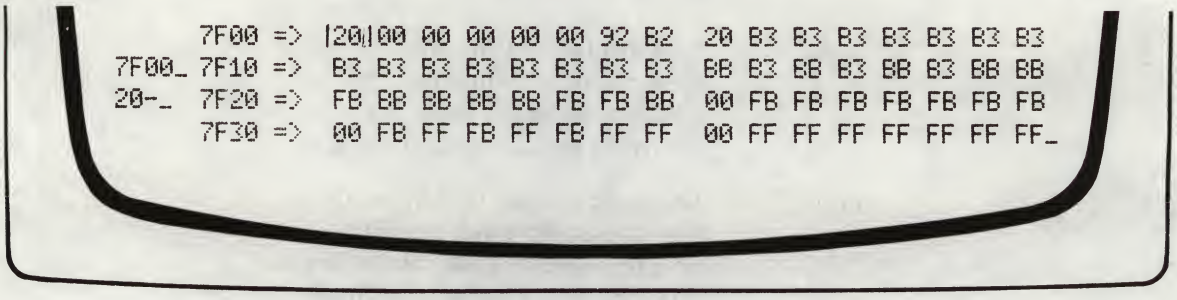
tells DEBUG to display the page of memory containing address 4080.

More on the M-command (modify memory)

Any time you wish to alter the contents of a memory location, type M*aaaa* and press the <SPACE>. This sets the memory modification address to *aaaa* and puts a memory modification prompt in the lower left corner of the Display. For example, typing

M7F00 <SPACE>

produces:



Note the vertical bars around the value of 7F00; These will appear wherever the modification address appears on the screen.

To modify the contents of 7F00, type the new, two-digit contents and press <SPACE>. The display will then be updated, and DEBUG will increment the modification address by one.

To leave an address contents unchanged, simply press <SPACE> without first entering a new contents. This will increment the modification address and leave the previous address unchanged.

To exit the modify memory mode, type X or **ENTER**.

If you simply type:

M <SPACE>

DEBUG will default to the last specified modification address, if any; otherwise 0000 will be used.

Frequently, two values on the display will be highlighted by vertical bars – one in the 64-byte memory display area, and another in the indirect memory area associated with the register pairs.

This is because the contents of the modification address happens to be displayed twice, one directly, one indirectly.

More on the G-command

To return to TRSDOS from DEBUG without re-initializing, type

`G402D` **ENTER**

DEBUG will then be re-entered under any of the three conditions noted above.

To disable DEBUG after using this exit, type

`DEBUG (OFF)` **ENTER**

`DIR` **ENTER**

To begin execution at the address in the PC register (while you're in the DEBUG mode), type

`G` **ENTER**

To reinitialize TRSDOS, type

`G0000` **ENTER**

More on the U-command (update display)

In the Update mode, only foreground tasks are executed. So to see anything happening, you need to look at registers or memory locations used by a foreground task.

The real-time clock makes a good example.

Type:

`D4040` <SPACE>

to display the values 4040 through 4046. These addresses store the time and date, as follows:

address	contents
4040	25mS real-time scheduling counter
4041	seconds
4042	minutes
4043	hours
4044	year
4045	day
4046	month

Now hit U and you'll see the values updated by the clock foreground task.

Other applications for DEBUG

DEBUG can be accessed via DISK BASIC, to help you locate stack pointers, table addresses, etc. See **DISK BASIC**.

DEBUG is also a handy way to create short object code programs, which can then be DUMPed onto diskette.

To disable DEBUG

As long as DEBUG is in the overlay area, TRSDOS may enter the debugging program unexpectedly, for example, upon an error. If you don't want this to happen, disable DEBUG by typing:

```
G402D ENTER          (to return to TRSDOS)
DEBUG (OFF) ENTER
DIR ENTER
```

TRACE (dynamic display of PC register)

TRACE[♭(*param*)]

where *param* = ON or OFF; ON is the default.

The TRACE command enables a foreground task which displays the contents of the user's program instruction counter (PC register) in the upper right of the Video Display. The 4-digit hexadecimal value will be updated every eight milliseconds with the current background program's execution address. For example:

```
TRACE ENTER
```

Since it is a foreground task, TRACE operates at all times – in DOS READY mode, DISK BASIC, or any other program. To temporarily disable TRACE, disable all interrupts (CMD"T" in DISK BASIC). When interrupts are re-enabled CMD"R" in DISK BASIC, TRACE will start up again.

Used with the DEBUG program, TRACE can be invaluable in debugging machine-language programs. It won't be of much use during BASIC program execution, though. To permanently stop TRACE, execute the command:

```
TRACE (OFF) ENTER
```

Library Commands

These commands are overlayed into the RAM area hex 5200-6FFF. They are loaded as requested in blocks; so, for example, DATE and TIME are both loaded when either is requested. TRSDOS will not waste time loading a command if the code is already in RAM.

AUTO (automatic key-in on power-up)

AUTO [*dos-command*]

where *dos-command* is a filespec for an operator command or an executable command file.

Note: To use AUTO, you must remove the write-protect tab from the system diskette.

The AUTO command lets you modify the power-up sequence, by specifying a command to be executed immediately after power-up.

Typing:

AUTO *dos-command* **ENTER**

causes TRSDOS to write *dos-command* as an "automatic key-in" on the drive 0 diskette, replacing any previous automatic key-ins. From that point on, every time you power up using that TRSDOS diskette, *dos-command* will be keyed in automatically whenever TRSDOS is initialized. An automatic key-in takes the place of keyboard input.

To restore the power-up sequence to normal, type:

AUTO **ENTER**

This will eliminate any automatic key-ins.

Examples:

AUTO CLOCK	on subsequent power-ups, the display clock command will automatically load and execute.
AUTO BASIC	on subsequent power-ups, TRSDOS will load DISK BASIC and begin the initialization dialog.

NOTE: You can override any automatic key-in by holding down the **ENTER** key during power-up. This may be your only way of regaining control of the system, for example, if *dos-command* is not a working command program.

ATTRIB (set protection attributes)

ATTRIB ♭*filespec*♭(*param*[,*param* . . .])

where *param* can be any of the following:

param	meaning
I	make file Invisible to normal Directory command
ACC=*psw1*	assign *psw1* as the new access password
UPD=*psw2*	assign *psw2* as the new update password
PROT=*level*	assign *level* as the new access protection level: (KILL, RENAME, WRITE, READ, EXEC)

The *filespec* must exist on one of the connected drives.

This command lets you alter the protection status of a file, by changing passwords and/or the degree of access granted by a password. (See **TRSDOS Overview**, "File Specifications" section.)

Specifying the I parameter gives the file the invisible attribute. To display Invisible files in the Directory, you have to specify the I parameter in the DIR command. There is no way to remove the I attribute, short of copying the file to a new file which does not have the I attribute.

Example:

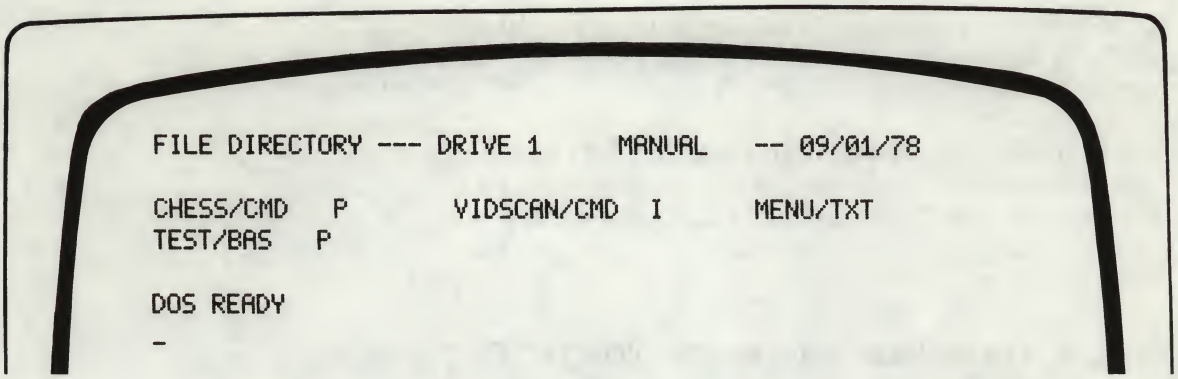
```
ATTRIB VIDSCAN/CMD:1 (I) ENTER

DOS READY
DIR :1 ENTER
```

```
FILE DIRECTORY --- DRIVE 1     MANUAL    -- 09/01/78

CHESS/CMD   P       MENU/TXT             TEST/BAS   P


DOS READY
DIR :1 (I) ENTER
```



All files are protected with two passwords, an access and an update password. Access and update passwords may be identical, and they may consist of all blanks. Use of the **update** password grants total privilege to a file – you can kill, rename, write, etc. Use of the **access** password, on the other hand, grants a limited privilege, as specified by a PROT parameter in the ATTRIB command.

The protection levels form a hierarchy, and each level implies access to all lower levels.

level	privilege
KILL	total privilege
RENAME	rename, write, read, execute
WRITE	write, read, execute
READ	read, execute
EXEC	execute only

When you create a file, the password you specify becomes both the access and the update password. (If you don't specify a password, a string of 8 blanks is assigned as a default password for both access and update.)

Once you have created the file, you can use ATTRIB to assign different values to the access and update passwords. Having two different passwords can be very useful in business applications.

For example, suppose you have a data file, PAYROLL, and you want an employee to use the file in preparing paychecks. Assume the file was created with default (blank) passwords.

Then:

```
ATTRIB PAYROLL (ACC=EMPLOYEE,UPD=MANAGER,PROT=READ)
```

would allow the EMPLOYEE to read the file, while only MANAGER could alter it.

To delete a password (set it to blanks), omit the password after the equals sign in the password specification. For example,

```
ATTRIB PAYROLL.MANAGER (ACC=)
```

sets the access password to blanks, and leaves the update password unchanged.

Note: To access a file from DISK BASIC requires a privilege of READ or higher.

CLOCK (display real-time clock)

CLOCK[ϕ(*param*)]

where *param*=ON or OFF; if no *param* is specified, ON is assumed.

Typing:

CLOCK **ENTER**

causes the internal real-time clock to be forcibly displayed on the top line of the Video Display (PRINT positions 53-60). Any characters present at those locations will be overwritten.

The clock display is updated once a second via a "foreground task". In other words, as long as the interrupts are enabled, TRSDOS will periodically interrupt whatever "background program" is executing (DISK BASIC, TAPEDISK, etc.), and update the clock display.

TRSDOS powers-up in a CLOCK OFF condition.

To stop the display-clock function, execute the command:

```
CLOCK (OFF) ENTER
```

See TIME command for information on the real-time clock.

COPY (make a duplicate file)

COPY␣*filespec1*␣TO␣*filespec2*

Creates a duplicate of *filespec1* under the new name *filespec2*. If filespec2 already exists, its previous contents are lost. The first file (*filespec1*) is unchanged by this command.

You must have at least two disk drives to copy a file from one diskette to another.

Examples:

```
COPY PAGE7/TXT:0 TO PAGE7/TXT:1
```

duplicates PAGE7/TXT on drive 0 onto drive 1, using the same name/extension.

```
COPY OLDFILE/BAS.PDQ TO DEADFILE
```

duplicates OLDFILE under the name DEADFILE. Note that OLDFILE is protected by a password, while DEADFILE is not. DEADFILE will be created on the first non write-protected drive in the sequence 0-3.

DATE (set date)

DATE␣mm/dd/yy

where mm is a 2-digit month specification, mm=01 to 12
dd is a 2-digit day specification, dd=01 to 31
yy is a 2-digit year specification, yy=00 to 99

For example, if it's August 3, 1978, type:

```
DATE 08/03/78 ENTER
```

This command resets the real-time date. At power-on, the date is set to 00/00/00. The date is updated each time the clock cycles through a 24-hour period. The real-time clock calendar includes the logic to account for 28, 29, 30 and 31-day months.

DEVICE

DEVICE

This command has no arguments or parameters. It simply lists all currently defined I/O devices: KI=keyboard, DO=video display, PR=line printer.

DEVICE **ENTER**

DIR (display directory)

Example:

DIR[♭:*d*] [♭(*param*[,*param* . . .])]

where :*d* = a drive specification, *d*=0,1,2 or 3, and 0 is the default
param = any of the following:

param	meaning
S	display all System and non-Invisible files
I	display all Invisible and non-System files
A	display disk space allocation for all files displayed

This command reads and displays the file directory of a specified or assumed drive. If no parameters are specified, only non-Invisible user files will be displayed.

Disk space allocation is indicated as follows: LRL (logical record length), EOF (end of file, i.e., highest record number used), and SIZE (measured in GRANules, where 1 granule = one-half track, or 1.25K bytes).

Examples:

DIR **ENTER**

displays all user files on drive 0. A typical output for this command might be:

```
FILE DIRECTORY --- DRIVE 0     TRSDOS   -- 10/03/78

VIDSCAN2/CMD        CLKAXESS/BAS        SELECTRC/DVR
TBUG/CMD            EDTASM/CMD          GLOSSARY/BAS
LISTER/BAS          TAPEDISK/CMD        KBFIX/CIM
DISKDUMP/BAS        GLOSSACC/BAS        VIDSCAN/CMD

DOS READY
_
```

```
DIR :1 (I,S) ENTER
```

displays all files, including System and Invisible files. A typical output for this command might be:

```
FILE DIRECTORY --- DRIVE 1     MANUAL    -- 09/01/78

BOOT/SYS SIP         DIR/SYS SIP          CHESS/CMD   P
MENU/TXT             TEST/BAS   P

DOS READY
-
```

Note the P beside some files. This indicates they have non-blank passwords.

```
DIR (A) ENTER
```

gives the disk space allocation on drive 0, user files only. Typically:

```
FILE DIRECTORY --- DRIVE 0     TRSDOS    -- 11/18/78

EDTASM/CMD          LRL=  256 / EOF=   27 / SIZE=    6 GRANS
RSM/CMD             LRL=  256 / EOF=   18 / SIZE=    4 GRANS
VHMTBUG/CMD         LRL=  256 / EOF=    0 / SIZE=    2 GRANS
SEQCHECK/TXT        LRL=  256 / EOF=    2 / SIZE=    1 GRANS
TBUG/CMD            LRL=  256 / EOF=    5 / SIZE=    2 GRANS
TAPEDISK/CMD        LRL=  256 / EOF=    2 / SIZE=    1 GRANS
CPRINT/BAS          LRL=  256 / EOF=    1 / SIZE=    1 GRANS
HMRSM/CMD           LRL=  256 / EOF=   18 / SIZE=    4 GRANS


DOS READY
-
```

If a Directory listing cannot fit on the screen, only the first 12 lines will be displayed. Press any key to see the remainder of the listing, in increments of 16 lines.

DUMP (dump memory to disk)

DUMP␣*filespec*␣(START=X'*aaaa*',END=X'*bbbb*'[,TRA=X'*cccc*'])

where *aaaa*, *bbbb*, *cccc* are 4-digit hexadecimal addresses

aaaa = starting point in RAM of the machine language program or data block to be dumped to disk; *aaaa* must be greater than 6FFF.

bbbb = ending point in RAM of the block; *bbbb* must be no smaller than *aaaa*

cccc = transfer address; when TRSDOS attempts to execute the file, it will start at *cccc*. If *cccc* is omitted, 402D will be used. This is the address of the normal re-entry into TRSDOS (i.e., re-entry with DOS READY displayed; no re-initialization).

If *filespec* already exists, its previous contents will be lost.

If *filespec* does not include an extension, TRSDOS will automatically assign the extension CIM (core image) to the file.

Once you have dumped a machine language program onto disk, there are two ways to execute it.

1) Simply type filespec ENTER . TRSDOS will load the file and begin execution at the transfer address.

2) Type DEBUG ENTER and then filespec ENTER After TRSDOS loads the file, it will enter the DEBUG package. PC will contain the transfer address. You can then single step the program (I command), call-step (C command), or execute it in full by typing:
G ENTER

Note: A file with the extension /CMD can be loaded and executed simply by typing the file name, without the extension, and pressing ENTER . TRSDOS will supply /CMD as a default extension.

Examples:

```
DUMP GRAPHICS (START=X'7000',END=X'70A0',TRA=X'7000')

DUMP DATA/CIM:1 (START=X'8000',END=X'8050')
```

KILL (delete a file)

KILL*b filespec*

This command deletes the specified file and frees the space for use by the system.

If no drivespec is included in the *filespec*, TRSDOS will search for the first drive which contains *filespec*, and attempt to delete that file. If the diskette is write-protected, TRSDOS cannot KILL the file.

Example:

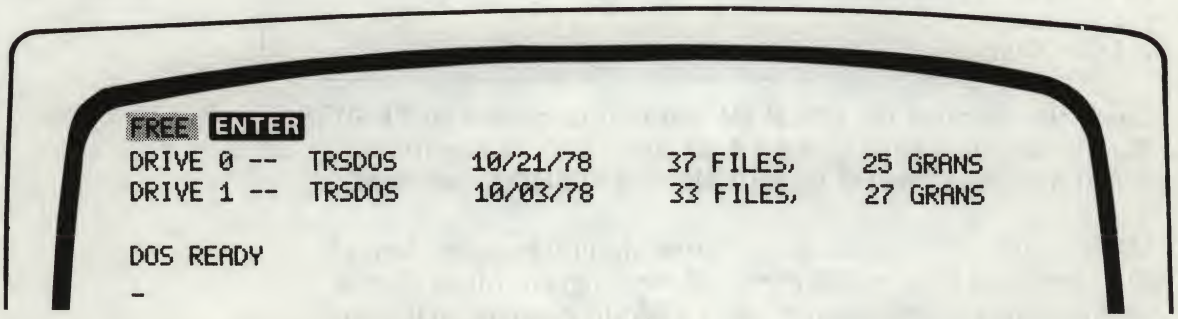
```
KILL OLDFILE/BAS.PASSWORD
```

FREE (display free space on all drives)

FREE

This command has no arguments or parameters. It displays the amount of free space remaining on all drives in use, in terms of files available and unused granules. (Each diskette can contain up to 48 user files; data diskettes have 67 granules available for user files; TRSDOS diskettes, 44 granules.)

For example:



LIB (display library commands)

LIB

Requires no arguments or parameters. This command displays all TRSDOS system library commands available. These are the commands which load between hexadecimal 5200 and 6FFF.

For example:

```
LIB ENTER
```

LIST (list text file contents to display)

LIST␣*filespec*

Reads the specified file and lists its contents on the Video Display. Because LIST gives an ASCII representation of the data in the file, *filespec* should refer to a text file. If you LIST a non-text file, the display will be filled with a meaningless sequence of ASCII and graphics characters.

Text files include:

- BASIC programs saved with the A option
- data files created by BASIC sequential write (PRINT#n) statements
- assembly language source code; etc.

To temporarily freeze the Display during LIST execution, hold down the SHIFT and @ keys until the listing pauses; press any key to resume execution. TRSDOS will only accept such a pause after listing a complete physical record – that's why you need to hold down the SHIFT @ keys until TRSDOS "notices" your pause command.

Example:

```
LIST PROG1/TXT
```

LOAD (load machine language file)

LOAD␣*filespec*

Loads the specified file into RAM and returns control to TRSDOS. The file specified must contain Z-80 object code, and normally would have been created by a DUMP or TAPEDISK command.

LOAD is useful for loading several programs into memory, so that all of them can then be called by a master program, which may be another machine language routine or a BASIC program. (Of course, all the different files must load into non-overlapping areas of RAM.)

To load subsidiary object code programs and then execute them via a master object code program, LOAD each of the subsidiary programs, then type the master filename and press **ENTER** .

Examples:

```
LOAD GRAPHICS
LOAD DATA/CIM:1
```

PRINT (list text file to line printer)

PRINT␣*filespec*

Works just like LIST, only the output is sent to the line printer. The file should be in text (ASCII) form.

Examples:

```
PRINT SEQCHEK/TXT
PRINT PAGE7/TXT:0
```

PROT (use diskette's master password)

PROT[␣:*d*][␣(*param*[,*param* . . .])]

where :*d* = a drivespec, d=0,1,2,3; if no drivespec is given the first drive is used

param can be any of the following:

param	meaning
PW	change Master Password
UNLOCK	remove passwords from all user files
LOCK	assign the master password to all user files

LOCK and UNLOCK are mutually exclusive; use only one.

This command changes the protection status of all non-System files on the specified drive. To use it, you need to know the diskette's Master Password, which is assigned during FORMAT or BACKUP. The diskette you reference must not be write-protected.

Note: Your TRSDOS diskette has the password, PASSWORD.

To change the Master Password, specify PW as a parameter. To remove passwords from all user files, specify UNLOCK. To place the diskette's Master Password on all user files, specify LOCK. (The Master Password then becomes the update and access password for those files.)

Examples:

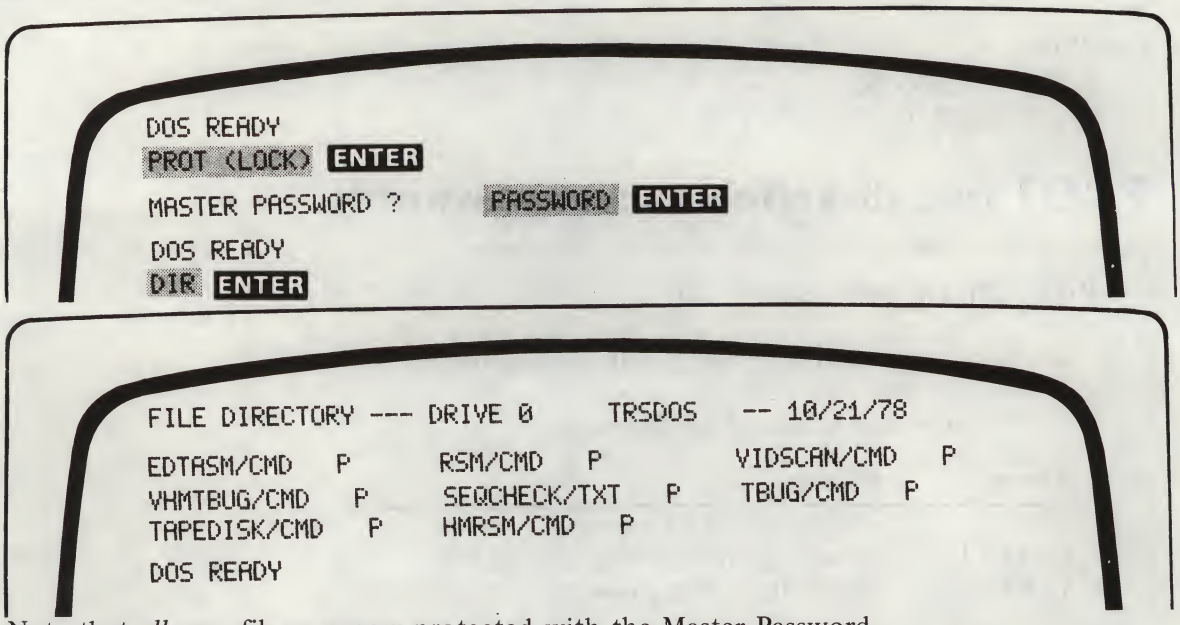
```
PROT :1 (UNLOCK) ENTER
```

After you enter this command, TRSDOS asks for the Master Password for the drive 1 diskette. If you enter the password correctly, TRSDOS will remove all user assigned passwords from files on that diskette.

```
PROT (PW,LOCK)
```

After you specify the Master Password correctly, TRSDOS will prompt you to enter a new Master Password. This new password will be assigned to all user files, since the command included the LOCK option.

A typical display sequence using the PROT command:



Note that *all* user files are now protected with the Master Password.

RENAME

RENAME␣*filename1*[/*ext1*][.*psw*][:*d*]␣TO␣*filename2*[/*ext2*]

where *filename1, filename2* are TRSDOS file names,
ext1,ext2 are extensions
:*d* is a drivespec (d=0,1,2,3)
psw is a password

This command changes a file's name from the first name/extension to the second name/extension. Note that the second name/extension should not include a password or a drivespec. The first file's specification may include a password and drivespec, as required to identify a desired file.

RENAME cannot be used to change a file's protection attributes or to move it to another drive. The previous passwords, protection level, and Directory attributes (Invisible for non-Invisible) will be assigned to the renamed file, and the file will remain on the same diskette.

RENAME also checks to see that the intended new name does not duplicate a filename currently on the same diskette. If it does, the command is cancelled and an error message is displayed.

Examples:

```
RENAME MATHPAK TO MATHPAK/BAS
```
adds an extension to the filename.

```
RENAME ABCDE/DAT TO ABCDEF/DAT
```
changes the file name only.

```
RENAME PAYROLL1/TXT.GSR TO PAYROLL2/TXT
```
changes the filename; the password is retained automatically.

```
RENAME FILE1:3 TO FILE2
```
changes the filename of the file on drive 3 only.

TIME (set real-time clock)

TIME␣*hh*:*mm*:*ss*

where *hh* is a 2-digit hours specification
mm is a 2-digit minutes specification
ss is a 2-digit seconds specification

This command sets the clock. On power-up, the clock is reset to 00:00:00.

Note: TRSDOS maintains a 24-hour/day clock format. After 23:59:59, the clock starts over at 00:00:00, and the day is incremented.

The current time is stored at locations hexadecimal 4040-4046; these values are updated via the realtime clock as long as interrupts are enabled.

Example:

```
TIME 08:24:00
```

See DATE and CLOCK

VERIFY (automatic read-after-write)

VERIFY[ϕ(*param*)]

where *param* = ON or OFF; ON is the default.

VERIFY **ENTER**

causes TRSDOS to verify all user disk writes (for example, file-writes from DISK BASIC). This will be useful when you want to be sure that no data is lost or altered during a disk write. For example, before you COPY a file, you may want to enable VERIFY.

However, when VERIFY is on, disk accesses are only about 50 percent as fast as normal.

Typing:

VERIFY (OFF) **ENTER**

disables the automatic read-after-write verification.
(note that TRSDOS powers up in a VERIFY (OFF) condition.)

Verify does not affect system table and directory writes; they are always verified.

Extended Utilities



TRSDOS

Contents of This Section

TRSDOS Utilities 2
BACKUP 2
FORMAT 4

Auxiliary Utilities 6
TAPEDISK 6
DISKDUMP/BAS 8

TRSDOS Utilities

These are special programs, not strictly a part of TRSDOS, which you can call to perform some very useful functions. Unlike system routines and library commands, these extended programs may use memory locations above hex address 6FFF; therefore any programs you have in RAM may be lost when you load a utility program.

BACKUP (duplicate a diskette)

BACKUP[ß:*d1*ßTOß:*d2*]

where :*d1* is a specification for the source drive
:*d2* is a specification for the destination drive
d1,*d2* = 0,1,2 or 3.

If you omit the drivespecs, BACKUP will prompt you to enter the source and destination drive numbers one at a time.

This utility duplicates an entire TRSDOS or data diskette. You can use any two drives for the backup, or you can perform the backup using drive 0, by swapping source and destination diskettes when BACKUP tells you to.

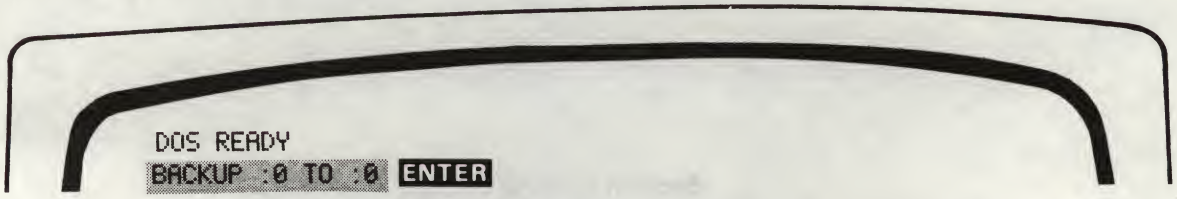
If the destination diskette is unformatted, BACKUP will format it, locking out any defective tracks, and will then proceed to copy all source disk files onto it. (If the destination disk cannot contain all the source disk data because of locked out tracks, the backup will be rejected.)

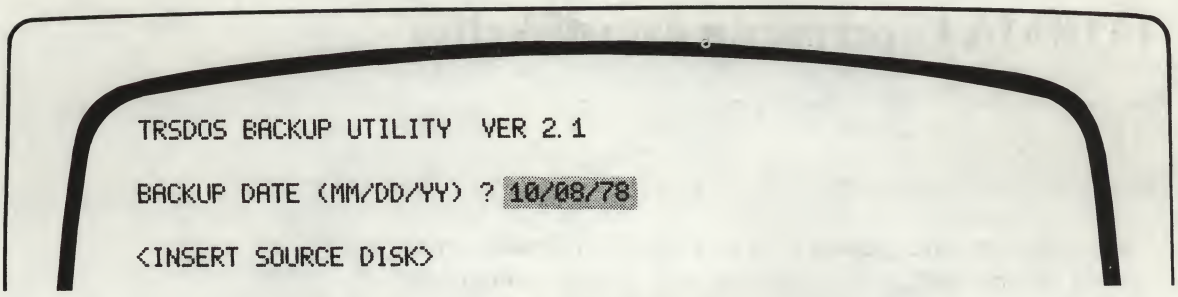
BACKUP will accept a pre-formatted diskette only when its Master Password and Diskette Name match that of the source disk. In this case, BACKUP will skip the formatting step and begin the copy and verify process. If for some reason, BACKUP rejects a diskette, erase the diskette with a bulk eraser and try again.

Examples:

```
BACKUP
BACKUP :0 TO :0
BACKUP :0 TO :1
```

Here's a typical BACKUP sequence, using only Drive 0.





BACKUP will then prompt you to insert source (original) and destination (duplicate) diskettes as necessary.

When using two drives for the BACKUP, you won't have to do any swapping.

IMPORTANT NOTICE

The BACKUP utility is provided solely for your personal use in maintaining safe copies of your TRSDOS and data diskettes. BACKUP automatically places copyrighted TRSDOS software onto each destination disk. TRSDOS users may BACKUP the system software solely for personal use.

See the Copyright Notice at the beginning of this Manual for more details.

FORMAT (prepare a data diskette)

FORMAT

This utility lets you prepare data diskettes containing a minimum of system information and leaving you with a maximum amount of space for program and data files. (TRSDOS diskettes have 44 granules/55K bytes available for your files; data diskettes, 67 granules/83.75K bytes.

Note: Data diskettes can only be used in drives 1,2, and 3, except during a BACKUP or FORMAT.

FORMAT takes a blank (new or magnetically erased) diskette, records track/sector boundaries on it, then initializes it with directory and bootstrap files. During the formatting process, TRSDOS will let you specify any tracks you'd like to lock out, so you can use them for non-TRSDOS files.

Unless you have another (non-TRSDOS) means of accessing the diskette, don't lock out any tracks.

FORMAT will lock out any defective tracks, to prevent data from being lost in these areas.

If you begin to get READ errors during accesses to a diskette, erase the diskette and re-format it. If there are defective tracks, FORMAT will lock them out, and you'll be left with an otherwise usable diskette.

To lock out tracks...

Specify them individually or as a range.

Example:
1,3-5 locks out tracks 1,3,4,5.

TRSDOS will never try to write to locked-out tracks.

Here is a typical FORMAT sequence, using Drive 1.

```
DOS READY
FORMAT ENTER
```

```
DISK FORMATTER UTILITY  2.1

WHICH DRIVE IS TO BE USED ? 1 ENTER
DISKETTE NAME ? MANUAL ENTER
CREATION DATE (MM/DD/YY) ? 10/08/78 ENTER
MASTER PASSWORD ? TRS80 ENTER
DO YOU WANT TO LOCK OUT ANY TRACKS? Y ENTER
WHICH TRACKS (0-34)? 1,6-8 ENTER
FORMAT THE LOCKED-OUT TRACKS? N ENTER
```

Auxiliary Utilities

TAPEDISK (copy tape file to disk file)

This utility lets you load a SYSTEM tape into RAM, and then dump it into a specified file on the disk. (SYSTEM tapes are created with the Editor/Assembler, TBUG, or supplied by Radio Shack.)

Do not attempt to use TAPEDISK to load tape files which load below hexadecimal address 54F4 (decimal 21748). TAPEDISK uses this area.

Note: Most Radio Shack SYSTEM tapes designed for use with LEVEL II TRS-80's will not work under DISK BASIC, because of differences in RAM usage under DISK BASIC and LEVEL II.

To load and execute TAPEDISK, type:

```
TAPEDISK ENTER
```

TAPEDISK will come up with the prompt,

```
?
```

Any time the prompt is displayed on the current line, you can enter one of the three TAPEDISK commands.

1) Load from tape

```
C
```

is the command to turn on the Recorder. (To use TAPEDISK, you should connect the recorder directly to the TRS-80 tape jack, not to the Expansion Interface jack.)

Type:

```
?C ENTER
```

When the file has loaded, you can load another SYSTEM tape, or enter another command.

2) Dump to disk

F␣*filename*[*/ext*] [*.password*] :*d*␣*aaaa*␣*bbbb*␣*cccc*

where *filename* is a TRSDOS filename
/ext is an optional extension;
.password is an optional password specification;
:*d* is a required drivespec, *d*=0,1,2 or 3;

aaaa is the hexadecimal starting address in RAM;
bbbb is the hex ending address in RAM;
cccc is the entry point for execution of the file.
All addresses are in 4-digit hexadecimal form.

When you're ready to dump the program from RAM onto disk, type in the F command. For example, if the program loaded into RAM addresses 7000-70FF, and the entry point is at 700A, you'd type:

```
?F USRCODE/CMD:1 7000 70FF 700A ENTER
```

After the dump, the prompt will return.

3) Exit to TRSDOS

E

This command returns you to TRSDOS, via the normal re-entry (no re-initialization).

Below is a typical TAPEDISK display sequence.

```
DOS READY

TAPEDISK ENTER

?C

?F GRAPHICS/DAT:0 6A00 6FFF 402D ENTER

?E ENTER

DOS READY
_
```

DISKDUMP/BAS (examine disk file)

This is a BASIC program. To execute it, you must load DISK BASIC first, and then load DISKDUMP/BAS:

```
BASIC ENTER
```

```
HOW MANY FILES? ENTER
MEMORY SIZE? ENTER
RADIO SHACK DISK BASIC VERSION 1.1
READY
RUN"DISKDUMP/BAS" ENTER
```

DISKDUMP lets you look at the contents of any of your disk files. It will help you experiment with various random and sequential disk output statements, and also help you to debug disk I/0 routines.

The program is written to dump to the Line Printer. If you do not have one connected, change all LPRINTs to PRINTs (lines 170,240,250) and change line 160 to:

```
160 GET1,SN
```

This program prompts you to enter the filename and then to enter the sector you want to examine. You can simply press ENTER without a number and the sector-by-sector examination will be sequential, starting with sector 1, the first physical record in the file.

If you specify a sector number higher than the EOF number (end-of file), no error message will be given and the "sector" will appear as zero-value bytes.

The sectors are printed 16 bytes at a time. These 16 bytes are displayed first in hexadecimal code, then with the corresponding ASCII code. The ASCII representation is surrounded by ! symbols. Periods are substituted for bytes which have no alphanumeric representation.

Below is a typical DISKDUMP session.

```
                SECTOR DUMP UTILITY 1.1

FILESPEC: SEQCHECK/TXT

SECTOR NUMBER (OR 'ENTER' FOR NEXT SECTOR): ENTER
```

```
FILESPEC: SEQCHECK/TXT              SECTOR:  1

  0    35 20 43 4C 53 3A 20 43    4C 45 41 52 20 31 30 30   !5 CLS: CLEAR 100!
 16    30 0D 31 30 20 41 24 3D    49 4E 4B 45 59 24 3A 49   !0.10 A$=INKEY$:I!
 32    46 41 24 3D 22 22 54 48    45 4E 31 30 0D 31 35 20   !FA$=""THEN10.15 !
 48    49 46 20 41 24 3D 22 40    22 54 48 45 4E 20 32 35   !IF A$="@"THEN 25!
 64    0D 32 30 20 50 52 49 4E    54 41 24 3B 3A 42 24 3D   !.20 PRINTA$;:B$=!
 80    42 24 2B 41 24 3A 47 4F    54 4F 31 30 0D 32 35 20   !B$+A$:GOTO10.25 !
 96    50 52 49 4E 54 3A 50 52    49 4E 54 22 44 41 54 41   !PRINT:PRINT"DATA!
112    20 49 4D 41 47 45 20 57    49 4C 4C 20 41 53 20 4F   ! IMAGE WILL AS O!
128    4E 20 4E 45 58 54 20 4C    49 4E 45 2E 20 28 22 43   !N NEXT LINE. ("C!
144    48 52 24 28 39 31 29 22    3D 42 59 54 45 20 44 45   !HR$(91)"=BYTE DE!
160    4C 49 4D 49 54 45 52 29    22 0D 33 30 20 46 4F 52   !LIMITER)".30 FOR!
176    49 25 3D 31 20 54 4F 20    4C 45 4E 28 42 24 29 3A   !I%=1 TO LEN(B$):!
192    20 50 52 49 4E 54 20 41    53 43 28 4D 49 44 24 28   ! PRINT ASC(MID$(!
208    42 24 2C 49 25 29 29 43    48 52 24 28 39 31 29 3B   !B$,I%))CHR$(91);!
224    3A 4E 45 58 54 0D 33 35    20 50 52 49 4E 54 0D 35   !:NEXT.35 PRINT.5!
240    30 20 4F 50 45 4E 22 4F    22 2C 31 2C 22 54 45 53   !0 OPEN"O",1,"TES!
```

TRSDOS Technical Information



TRSDOS

Contents of This Section

Memory Organization 2
Disk Organization 2
File Structure 3
System Routines for Assembly I/O 5
Data/Device Control Blocks 6
Physical and Logical Records 7
Fundamental TRSDOS I/O Calls 8
TRSDOS Error Codes/Messages 12

Memory Organization

The TRS-80 Disk Operating System is comprised of 1K of ROM resident CIO (Character-oriented I/O) drivers and 4K of RAM drivers, schedulers, tables, pointers, etc. The ROM resident CIO drivers are also used by LEVEL II BASIC and therefore are part of its 12K ROM requirement.

Since LEVEL II is upward compatible with DISK BASIC, an additional 0.5K of RAM is required for both versions of BASIC. This means that user memory starts at hex 5200, resulting in 11.5K of user RAM in a 16K machine.

Note: The memory which is **completely** untouched by both TRSDOS and DISK BASIC code begins at hex 7000.

TRSDOS is comprised of a resident system and several overlays which are loaded from disk as the need arises (for example, to open or close a file).

The system has a modular design. System entry-point vectors are in the lowest portion of the 4K RAM, followed by the interrupt handling, disk file handling, task scheduling and general purpose resident system routines. System buffers and overlays comprise the last portion of the 4K RAM requirement.

Since all major system commands are actually loaded as needed from disk in the form of utilities (the "library commands" and the extended utility programs), the TRSDOS system facilities can easily be enhanced without affecting the RAM memory requirement.

Disk Organization

Each TRSDOS system diskette contains a TRSDOS system, a utility command library, a file directory, and system tables.

The minimum system overhead amounts to one full track of directory information and a half track of TRSDOS bootstrap program and other information. This means that every TRSDOS diskette is self-loading, although it may or may not actually contain the TRSDOS system. This is done to prevent the Computer from attempting to bootstrap a diskette containing only user data files.

The utility command library is optionally available on the diskette. Since the utility command programs are not always required, it will often be advantageous for multi-drive users to format diskettes for use in drives 1 through 3. Such "data diskettes" contain a minimum of system code, leaving more space for user

files. Maximum file size is limited only by the physical size of the diskette, since a file must be wholly contained on one diskette.

Each diskette is single-sided and has 35 tracks of information. Each track contains 10 sectors of 256 bytes each. See **Mini Disk Operation**, "How a Diskette Works".

Normally, data read/write operations may only be initiated at sector boundaries, and must consist of exactly 256 bytes. However, TRSDOS allows the user to have maximum flexibility with minimal effort by automatically blocking and de-blocking all file accesses to user-specified logical record lengths, even if this requires "spanning" of two sectors.

The system disk file structure allows maximum use of disk file space by automatically segmenting files across a diskette in several small pieces. These pieces are correlated into one logically contiguous file by the system without your needing to know the physical file location. This structure eliminates time-consuming disk-packing operations.

File Structure

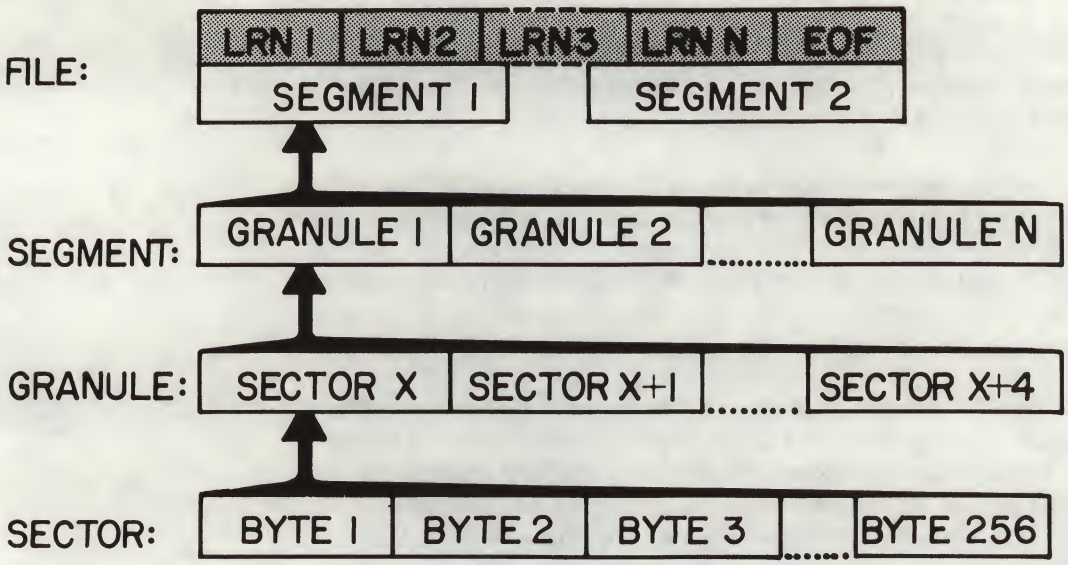
A TRSDOS file is composed of one or more segments of storage space. Each segment consists of from one to 32 physically contiguous granules of storage. A granule is the minimum allocatable unit of storage, and consists of five sectors (1.25K bytes). (See Figure below).

Since a file is always lengthened by granules, a small amount of free storage is generally present at the end of every file. This free storage allows minor file additions to be made in space which is physically contiguous to the file.

The effect is to decrease the amount of "thrashing" present in a file which has had frequent additions made. (A wholly sector-mapped system could not offer this benefit.)

Every time a disk file is extended (either initialized or lengthened), extra granules may be allocated to that file, depending on the file's accumulated length, diskette space, saturation, etc. These extra granules, along with all granules after the one containing the file's EOF mark, are recovered and returned to the system when the file is closed.

A TRSDOS file



LRN: Logical Record Number, used to specify an individual, user-defined logical record. Such a logical record is the smallest unit of information which can be addressed during disk input/output (a physical record is the unit which is actually read from or written to disk).

File: A group of logical records; the largest unit of information which can be addressed by a TRSDOS command.

Sector: A physical record, composed of 256 contiguous bytes.

Granule: The minimum allocatable unit of storage for a particular file.

System Routines for Assembly-Language I/O

This information is provided for customers who wish to write their own assembly level I/O routines. An explanation of the calling sequence and parameters for each necessary I/O routine is given. A knowledge of Z-80 machine code is assumed.

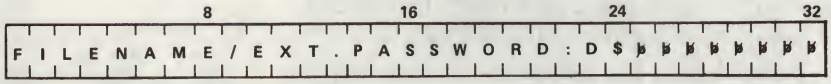
The following notations are standard in this section:

HL=> *xxxx*	Registers HL contain the address of (point to) *xxxx* in machine format. (If address of *xxxx*=34B2H then the values in the registers are: H=34 ; L=B2)
DE=> *xxxx*	Registers DE contain the address of (point to) *xxxx* in machine format. (If address of *xxxx*=5AF1H then the values in the registers are: D=5A ; E=F1)
B= *xx*	Register B contains the numeric value of *xx* in binary form. If *xx*=64 decimal, then B=40H.
A= *xx*	Register A contains the numeric value of *xx* in binary form. If *xx*=127 decimal, then A=7FH. Register A is used to return the TRSDOS error code for I/O calls. A complete list of error codes and their meanings appears at the end of this chapter.
Z=OK	Zero flag is set (OK) if successful return from the system routines.
X'*nnnn*'	Hard RAM address in hex notation (e.g., 402D is X'402D').
LRL	Logical Record Length. 1-255 bytes only. You can define records any length you wish up to 255 bytes maximum. A length of zero is a special case for physical records only, and indicates the LRL=256 bytes.
BUFFER	256 user designated bytes in RAM for TRSDOS to read sectors from or write sectors into. If LRL=0, this area is the responsibility of the user to manage before and after I/O. TRSDOS manages this area if LRL is between 1 and 255 bytes. Do not alter this area when using logical record processing.
UREC	User record: the address of the contiguous RAM byte-string assigned by the user as his logical record area. Its length must be equal to LRL. It is a different area from BUFFER.

DCB before OPEN and after CLOSE:

The DCB is defined as 32 contiguous bytes of RAM designated by the user. Before OPEN and after CLOSE, it is a left justified, compressed (no spaces) ASCII string, as in a standard TRSDOS filespec:

CONTENTS OF 32 - BYTE DCB



Notes: /EXT, .PASSWORD, :D are optional.
$ stands for a carriage return (X'OD')
ß stands for a blank (X'20')

Explanation of DCB while OPEN:

lsb/msb is least significant byte followed by most significant byte in Z80 RAM format (i.e. addr=7CC8 in RAM is C8 7C).

Addr.	Len.	Explanation
DCB+0	3	Reserved
+3	2	Physical Buffer address (lsb/msb)
+5	1	Offset to delimiter at end of current record
+6	1	File drive number residence
+7	1	Reserved
+8	1	EOF offset of last delimiter in last physical record
+9	1	LRL (logical record length)
+10	2	NRN (next record no. – open sets=X'0000' – lsb/msb)
+12	2	ERN (ending record no. – last in file – lsb/msb)
+14	18	Reserved

NRN Next Record Number defines which record is to be read or written by the next system call for READ or WRITE. It is automatically incremented by one after each system call. In order to process random files, use the POSN call to direct TRSDOS to the record you wish to transfer next.

ERN Ending Record Number is the last record number currently in the file. It is put into the directory at CLOSE time, so if it is expected to be correct, the user *must* close his files after adding records to a file. This value may also be used to position to end of file so that new records may be added to the end of the file. To position to the end of file use a call to POSN with a record number of ERN+1. POSN is described later.

Physical and Logical Records in TRSDOS

A physical record is defined as one sector of disk. One sector of disk contains 256 user data bytes. The artificial term "granule" is defined to be 5 sectors of disk space. There are 2 granules on each of the 35 tracks on the disk. A granule is the least amount of space allocated by TRSDOS. For programming purposes, the physical records in a file are numbered from 0 to N. The largest record number (N) in a file will then be five times the number of granules allocated minus one ((5*G)–1). All TRSDOS granule allocations are made as needed **at the time of write, not when the file is created.**

Bytes	Sectors	Granules	Tracks	Disk
256	1	–	–	–
1280	5	1	–	–
2560	10	2	1	–
89600	350	70	35	1

Disk Space Table : For each 5-1/4" Disk Drive

A logical record is defined by the user of TRSDOS. It may be anywhere from 1 to 255 bytes in length. Once a file is opened with a specific LRL (Logical Record Length), the length is fixed until the file is closed. To change a file's LRL, you must CLOSE it and re-OPEN it with the new LRL.

Each opening of the file sets a single, fixed record-length. TRSDOS will "block" logical records into (or from) one physical record for maximum space utilization on the disk.

Blocking is putting more than one logical record into one physical record. For instance, four 64-byte logical records will fit into one 256-byte physical record. A logical record may be broken into two parts by TRSDOS in order to fill the last portion of one physical record entirely before beginning to use the next physical record (i.e. records are spanned). This occurs when the physical record length is not an even multiple of the logical record length.

If the user wishes to do his own blocking, he may specify a logical record length of 0 bytes at the time of INIT/OPEN and must himself manage the contents of the physical record buffer area of 256 bytes. TRSDOS **will not move** a logical record for the user if LRL=0; in this particular case it will only read/write the physical record to/from the buffer.

Fundamental TRSDOS I/O Calls

There are eight fundamental TRSDOS routines involved in handling file I/O. These are:

INIT	Creates a new file in the directory and opens it. No granule allocation is done.
OPEN	Opens an existing disk file.
POSN	Position for reading/writing a particular logical record.
READ	Reads one logical record into RAM from disk or from the physical buffer.
WRITE	Writes one logical record from RAM onto disk or into the physical buffer.
VERF	Writes then verifies by reading back and comparing to the original data written from RAM. Only pertains to LRL=0 physical records.
CLOSE	Closes an open file.
KILL	Closes a file and erases it from the directory.

The detailed calling sequences and discussions for each of these routines follow. Note that **all** of these system calls use register F and do not restore its value before return. In order to properly apply this data, you should read through all of these descriptions and clear up all of the points that are not obvious to you by using other reference materials. If you are successful in doing this you will find that TRSDOS is a workable tool for your programming ideas. The jump vectors supplied here and the descriptions especially pertain to TRSDOS Version 2.1 only. Future releases of TRSDOS may alter some of these descriptions or addresses.

INIT (jump vector = X'4420')

INIT is provided as an entry point to TRSDOS which will create a new file entry in the directory and open the DCB for this file. INIT scans the directory for the filespec name given in the DCB. If the filespec name is found, INIT simply opens the file for use. If the name is not found, a new file is created with the filespec name.

entry: HL= >BUFFER (see beginning of this section for notation)
DE= >DCB
B= LRL
CALL 4420H

exit: Z=OK
C carry flag is ON if a new file was created
A=TRSDOS error code. (Error codes listed at end of this chapter)

OPEN (jump vector = X'4424')

OPEN provides a way to open the DCB of a file which already exists in the directory. The DCB **must** contain the filespec of the file to be opened **before** entry to OPEN.

entry: HL=> BUFFER
DE=> DCB
B= LRL
CALL 4424H
exit: Z=OK
Z=0 if file does not exist.
A=TRSDOS error code.

POSN (jump vector = X'4442')

POSN positions a file to read or write a randomly selected logical record. Since it deals with logical records, the proper computation is done to locate which physical record(s) contain the data. Following a POSN with a READ or WRITE will transfer the record to/from RAM.

Note that positioning to logical record zero sets the file to read the first logical record in the file. To position to end of file in order to add new records onto the end, use the record number ERN+1 (see page 2).

entry: DE=> DCB (must have been opened previously)
BC= Logical record number to position for.
CALL 4442H
exit: Z=OK
A=TRSDOS error code.

READ (jump vector = X'4436')

If LRL>0, READ transfers the logical record whose number is in the DCB as NRN (see page 2) into the RAM area addressed as UREC for the length LRL as defined at open time. The record comes from the RAM BUFFER defined at open time. If TRSDOS must read a new physical record to satisfy the request, it will do so. "Spanned" logical records will be re-assembled as necessary. READ automatically increments NRN by one in the DCB after the transfer is completed. INIT/OPEN sets NRN=X'0000' in order to read the first record with the first READ.

If LRL=0, READ transfers one physical record into the RAM BUFFER, which was defined at open time, from the disk file. Registers HL are ignored. READ increments NRN as above.

entry: HL= > UREC if LRL is not zero. Unused if LRL=0.
DE= > DCB
CALL 44364
exit: Z=OK
A=TRSDOS error code. (EOF=X'1C' or X'1D')
(see errors 28,29 for EOF or NRF)

WRITE (jump vector = X'4439')

IF LRL>0, WRITE transfers the one logical record from the RAM area addressed as UREC for the length LRL as defined at open time. The record goes into the RAM BUFFER which was defined at open time. If TRSDOS must write a physical record in order to satisfy the request, it will do so. "Spanning" will be handled by TRSDOS as necessary. At INIT/OPEN time the DCB value of NRN is set to X'0000' so that the first record will be written. After each logical record is transferred, the NRN value in the DCB will be incremented by one.

IF LRL=0, WRITE transfers one physical record from the RAM BUFFER into the disk file using the NRN in the DCB. BUFFER IS DEFINED at INIT/OPEN time only. The DCB value NRN is updated as above, after the WRITE.

entry: HL= > UREC if LRL is not zero. Unused if LRL= 0
DE= > DCB
CALL 4439H
exit: Z=OK
A=TRSDOS error code.

VERF (jump vector = X'443C')

The only difference between VERF and WRITE is that VERF writes one physical record to disk and then reads it back into a special TRSDOS RAM area not defined by the user. This special area and the original write buffer are then compared byte by byte to assure that the record was successfully written.

entry: HL= > Same as WRITE above.
DE= > DCB
CALL 443CH
exit: Z=OK
A=TRSDOS error code.

CLOSE (jump vector = X'4428')

CLOSE closes a file from the last processing done. **It is very important to do a CLOSE on every file opened before the program ends.** If you do not close a file, the directory entry for this file is incorrect if any new records have been written into the file. Other cases are not given here, but it is very important to TRSDOS that all of the "housekeeping" is complete for file management.

entry: DE=> DCB
CALL 4428H
exit: Z=OK
A=TRSDOS error code.

KILL (jump vector = X'442C')

KILL deletes the directory entry for an open file and then completes the close on the DCB. The disk space released by the old file is now re-useable for other purposes. Otherwise KILL is the same as CLOSE.

entry: DE=> DCB
CALL 442CH
exit: Z=OK
A=TRSDOS error code.

Supplementary Information

Other routines and addresses which may be of interest are defined here. Pay particular attention to the error routine. It does NOT perform error recovery. It displays TRSDOS error messages on the video display.

(1) CALL 402DH – Normal return to TRSDOS at program end.

(2) X'4318': address of the 64-byte buffer that contains the last TRSDOS command that was entered. Useful to decode special parameters entered when program was executed (run).

(3) If HL => 8-byte buffer, then:
CALL 446DH returns the time of day into the 8 bytes in the ASCII format – HH:MM:SS
CALL 4470H returns the date into the 8 bytes in the ASCII format – MM/DD/YY

Binary forms of the time and date are located in TRSDOS RAM at these locations:
X'4040' clock – real time clock heartbeat count. 25ms.
X'4041' time – binary – 3 bytes – sec,min,hrs
X'4044' date – binary – 3 bytes – yr, day, mon

(4) Printing TRSDOS error codes on the video display.

```
        CALL  4420H   Example of system I/O call.  Any call
                      is ok.  Zero flag not set means an error
                      has occurred during the I/O attempt.
        JR    Z,OKGO  Ignore error message display if no
                      error.
        OR    80H     Optional for detailed error message.
                      Register A already contains proper
                      code for a single line message display.
        CALL  4409H   Display error message on video screen.

              Optional user error recovery code goes here

        OKGO continue with program here - - -
```

TRSDOS Error Codes – Returned in Register A

decimal number	prob. causes*	error description
00	–	No error
01	MD	Parity error during header read
02	D	Seek error during read
03	XK	Lost data during read
04	MD	Parity error during read
05	FMD	Data record not found during read
06	P	Attempted to read system data record
07	P	Attempted to read system data record
08	UP	Device not available
09	MD	Parity error during header write
10	D	Seek error during write
11	XC	Lost data during write
12	MD	Parity error during write
13	FMD	Data record not found during write
14	XD	Write fault on disk drive
15	UDX	Write protected diskette
16	PS	Illegal logical file number (dcb bad)
17	MPDS	Directory read error
18	MPDS	Directory write error
19	UP	Illegal file name (dcb bad)
20	MPDS	GAT read error (Granule Allocation Table)
21	MPDS	GAT write error
22	MPDS	HIT read error (Hash Index Table)
23	MPDS	HIT write error
24	UP	File not in directory
25	UP	File access denied (protection violation)

*See Explanation, next page.

decimal number	prob. causes	error description
26	UP	Directory space full (48 files max)
27	UP	Disk space full (70 granules max)
28	P	EOF encountered (End Of File)
29	P	NRF (No Record Found) out of file range
30	UP	Full directory. File can't be extended.
31	UP	Program not found
32	UP	Illegal drive number specified
33	UP	No device space available for new device
34	MPUS	Load file format error. Not a program.
35	XCS	Memory fault
36	PUXC	Attempted to load ROM memory
37	P	Illegal access attempted to protected file
38	UP	File has not been opened
39-62		Not defined yet. Reserved
63	P	Unknown error code

Explanation of probable cause codes: (column 2)

C = TRS80 CPU fault
D = Disk drive fault
F = Diskette not formatted
M = Diskette media fault
P = User program error
S = TRSDOS fault. Reboot
U = User procedural error
X = Expansion Interface fault

DISK BASIC



LANGUAGES

Contents of This Section

Introduction 2
Enhancements to LEVEL II 5
Disk Features 26
File Manipulation 28
File Access 33
Sequential Access Techniques 60
Random Access Techniques 65
DISK BASIC Error Messages 77

Introduction

DISK BASIC is a set of enhancements to LEVEL II BASIC, plus features to allow disk input/output of BASIC programs and data. It is a memory image file stored on the TRSDOS software diskette with the name BASIC and extension /CMD.

When DISK BASIC is loaded into RAM, it automatically takes control of the LEVEL II BASIC ROM program, using almost all of its routines and adding others. This is possible because LEVEL II was designed with upward compatibility built-in. (You can see this by examining the memory map for LEVEL II, in particular, hex addresses 37DE-37EC.)

When loaded, DISK BASIC occupies approximately 5.8K bytes of RAM, beginning at hex address 5200 (decimal 20992).

To load and execute DISK BASIC, first power-up the Disk Operating System (see System Operation), so that

```
DOS READY
```

is displayed. Now type:

```
BASIC ENTER
```

TRSDOS will load BASIC into RAM, and BASIC will begin the "initialization dialog". This is a series of questions and answers which tell BASIC how to organize memory according to your needs.

The first question is,

```
HOW MANY FILES?_
```

You repond with the maximum number of disk files that will be open (in use) at any one time – any number from zero to 15.

(Every program or data set you store on the disk is referred to as a "file". In fact, everything on the disk, including system software, exists in the form of files.)

The number you enter tells BASIC how many disk I/O **buffers** and **data control blocks** to create (for definitions, see **Glossary**). If *n* files are to be in use at once, then *n* buffers will be required. Each buffer will take 290 bytes from your available RAM (256 for the buffer plus 34 for a data control block [DCB]), so don't enter an unnecessarily large number.

If you simply press ENTER without entering a number, BASIC will use a default value of 3; so you'll be able to have 3 file buffers in use at once.

Note: DISK BASIC automatically creates a buffer for loading, saving and merging BASIC programs. This buffer exists in RAM below any data file buffers you may request. It is always available for program I/O, regardless of how you answer the FILES? question.

Suppose you're going to be using 2 files: 1 for inputting data, 1 for outputting data. Then you might answer 2 to the FILES? question. However, if only 1 of these files will be open at once, then you really only need to reserve 1 file buffer/control block.

Examples:

```
HOW MANY FILES? 0 ENTER
```

causes BASIC to set aside zero buffers for I/O to disk files. You won't be able to open files, but you will have the maximum amount of RAM for use by your program.

```
HOW MANY FILES? 15 ENTER
```

tells BASIC to create 15 I/O buffers and control blocks; you will then be able to have 15 files open at once; however, this will reduce your available memory by 15*290 = 4350 bytes.

```
HOW MANY FILES? ENTER
```

tells BASIC to use a default of 3 for the number of files to be in use at once.

After you answer the FILES question, BASIC will ask:

```
MEMORY SIZE?_
```

You respond with the highest memory address (in decimal form) you want BASIC to use for storing and executing your BASIC programs. Addresses above the number you specify will then be protected from use by BASIC.

Here's why you might want to protect memory:
You can load machine-language programs or routines into high memory, and then access these routines from DISK BASIC via specially defined USR*n* functions, or via the SYSTEM command. These machine language routines may be loaded from tape using the SYSTEM command, LOADed in the DOS READY mode, or placed in memory one byte at a time using either DEBUG or BASIC POKE commands. If you do not reserve memory, such routines will be destroyed during execution of BASIC statements.

Example:

```
MEMORY SIZE? 32000 ENTER
```

causes BASIC to protect addresses above 32000. If you have 16K of RAM, this means that you'll have 32767-32000= 767 bytes protected for storing your machine language routines.

If you don't want to reserve any memory, just press **ENTER** without typing a number.

```
MEMORY SIZE? ENTER
```

You will then have the maximum amount of RAM available for use by BASIC.

Refer to the Memory Map for decimal addresses of the various TRS-80 memory configurations (16K, 32K, 48K).

After you answer the MEMORY SIZE question,

```
RADIO SHACK DISK BASIC VERSION 1.1
READY
>_
```

will be displayed. You are now operating under DISK BASIC.

To exit BASIC and return to the DOS READY mode, type:

```
CMD"S" ENTER
```

This results in a normal return to DOS – without re-initialization of the system. If you have a BASIC program in RAM, it will be lost, so be sure to save it on disk or tape before using CMD"S".

Enhancements to LEVEL II BASIC

DISK BASIC adds many features to LEVEL II which are not disk-related. They are listed below along with abbreviated descriptions. Detailed descriptions follow in alphabetical order.

&H	Hexadecimal-constant prefix
&O	Octal-constant prefix
CMD"D"	Enable and load the real-time debugging program
CMD"R"	Enable interrupts (start real-time clock)
CMD"S"	Normal return to TRSDOS (jump to EXIT routine)
CMD"T"	Disable interrupts (turn off real-time clock)
DEF FN	Define an implicit BASIC-statement function
DEF USR	Define the entry point for an external machine-language routine
INSTR	Instring function; find substring in target string
LINE INPUT	Input a line from keyboard
MID$=	Replace portion of target string (used on left of equals sign)
TIME$	Get time and date from real-time clock
USR*n*	Call external routine (*n*=0,1,2,...,9)

Cassette Operations

Before any BASIC cassette input or output operation, you must disable interrupts with the CMD"T" command. This is because such cassette operations are timing-sensitive and cannot work if they are being interrupted every 25 milliseconds. When the cassette operation is complete, you can re-enable interrupts by executing the statement CMD"R".

CLOAD allows no filename in DISK BASIC. Therefore you cannot use such a filename to sort through several tape files. CLOAD will always load the first file encountered on the tape. CSAVE, however, still requires the filename. This way, programs CSAVEd under DISK BASIC can be loaded and sorted through via the LEVEL II CLOAD"*filename*" command.

CLOAD? (CLOAD-verify), used in LEVEL II to compare a BASIC program in RAM with one on tape, will not work with programs saved on tape under LEVEL II. **It will work with programs saved under DISK BASIC.**

Error Messages

When an error occurs, DISK BASIC "spells out" the full error message, not just the abbreviation. This saves you from having to look it up.

Example:

```
ERROR(14) ENTER
```

DISK BASIC responds with:

```
OUT OF STRING SPACE
```

Note: The ERROR function, used to simulate error conditions, will work only with non-disk error codes.

&H and &O (hex and octal constants)

Often it is convenient to use hex (base 16) or octal (base 8) constants rather than their decimal counterparts. For example, memory addresses and byte values are easier to manipulate in hex form. &H and &O let you introduce such constants into your program.

&H and &O are used as prefixes for the numerals that immediately follow them:

&H*dddd*	
where *dddd*	is a 1 to 4 digit sequence composed of hexadecimal numerals 0,1,...9,A,B,...,F.
&O*ddddd*	
where *ddddd*	is a sequence of octal numerals 0,1,...,7. and &O*ddddd*< =177777 decimal. **Note:** The O can be omitted from the prefix &O. Therefore &O*ddddd*=&*ddddd*.

The constants always represent signed integers.
Therefore any hex number greater than &H7FFF, or any octal number greater than &O77777, will be interpreted as a negative quantity. The following table illustrates this:

Octal	Hex	Decimal
&1	&H1	1
&2	&H2	2
&77777	&H7FFF	32767
&100000	&H8000	–32768
&100001	&H8001	–32767
&100002	&H8002	–32766
&177776	&HFFFE	–2
&177777	&HFFFF	–1

Hex and octal constants cannot be typed in as responses to an INPUT prompt or be contained in a DATA statement. Often the hex or octal constant must be enclosed in parentheses to prevent a syntax error from occurring.

Examples:

```
PRINT &H5200,&O51000
```

prints the decimal equivalent of the two constants (both equal 20992).

```
POKE &H3C00,42
```

puts decimal 42 (ASCII code for an asterisk) into video memory address hex 3C00.

```
100 FOR I=(&H3C00) TO (&H3FFF) STEP (&H40)
200 IF A=(&H37E8) THEN A=A+1
300 POKE A%, (X% AND &HFF)
```

Masks the most significant byte of X% and POKEs the result into location A%.

CMD"D" (execute DEBUG program)

CMD"D"

Executing this statement causes the TRSDOS debugging program to load and execute. (See TRSDOS Commands, DEBUG.) Your BASIC program is unaffected, since DEBUG loads below DISK BASIC.

To return to BASIC without re-initialization, type

G **ENTER**

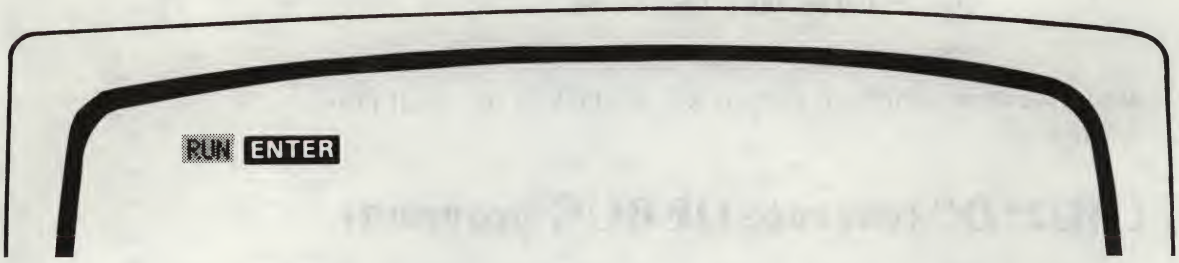
The READY message will appear and you can continue in BASIC.

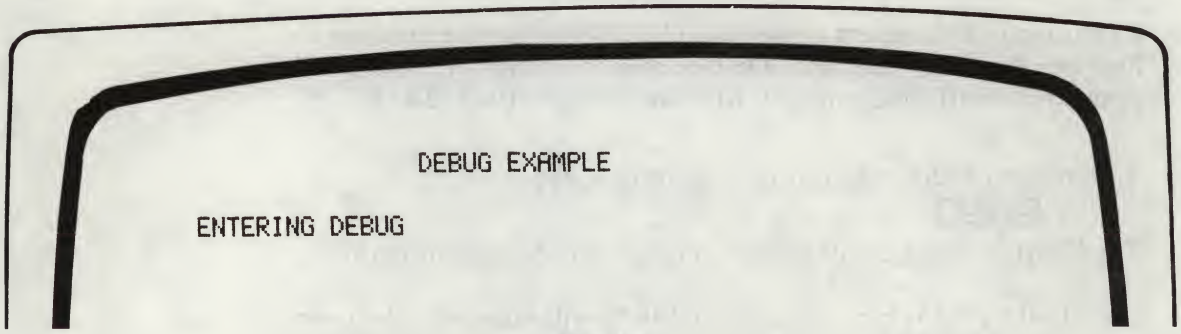
Once CMD"D" has been executed, DEBUG will take over whenever you press the BREAK key. Pressing G **ENTER** will return you to BASIC again. Type CONT to continue any program that was executing when you typed BREAK.

To return from DEBUG to the BASIC initialization sequence, type G5200 **ENTER** . You will lose any BASIC program text or variable values.

Examples:

```
100 '          PROGRAM: DEBUG
110 ' EXAMPLE OF EXECUTION WITH DEBUG WITHIN A PROGRAM
120 '
130 CLS: PRINT TAB(15); "DEBUG EXAMPLE": PRINT
140 PRINT"ENTERING DEBUG"
150 FOR I=1 TO 500: NEXT I    'DELAY A WHILE
160 '
170 ' *** ENTER DEBUGGING PACKAGE ***
180 '
190 CMD"D"
200 '
210 ' *** RETURN HERE WHEN "G" ENTER TYPED IN DEBUG ***
220 '
230 CLS: PRINT: PRINT "YOU HAVE RETURNED FROM DEBUG"
240 END
```





```
AF = 44 42 -Z----N-
BC = 69 01 =>  57 49 54 48 49 4E 20 41  20 50 52 4F 47 52 41 4D
DE = 69 B8 =>  44 22 00 C3 69 C8 00 3A  93 FB 00 FD 69 D2 00 3A
HL = 40 B7 =>  69 55 FF FF FF FF FF FF  FF 00 00 00 00 00 00 54
AF'= FF FF S21H1PNC
BC'= 4D BE =>  51 51 CD FC 51 7E 23 18  EC 02 02 00 4E 03 32 E7
DE'= 01 07 =>  4D 4F 52 59 20 53 49 5A  45 00 52 41 44 49 4F 20
HL'= 4D 00 =>  F2 51 06 10 CD 65 51 3A  5D 40 FE 41 20 13 CD F2
IX = 40 15 =>  01 E3 03 00 00 00 4B 49  07 58 04 31 3E 00 44 4F
IY = FF FF =>  FF F3 AF C3 74 06 C3 00  40 C3 00 40 E1 E9 C3 9F
SP = BD 6C =>  BA 69 1E 1D 00 00 04 04  20 00 00 00 00 00 00 00
PC = 57 08 =>  E1 C9 3A 29 5B F6 C0 CD  09 44 E1 C9 D7 E5 3E 11
     1010 =>  28 10 FE 44 28 0C FE 30  28 F0 FE 2C 28 EC FE 2E
     1020 =>  20 03 2B 36 30 7B E6 10  28 03 2B 36 24 7B E6 04
     1030 =>  C0 2B 70 C9 32 D8 40 21  30 41 36 20 C9 FE 05 E5
G ENTER 1040 =>  DE 00 17 57 14 CD 01 12  01 00 03 82 FA 57 10 14
```

```
YOU HAVE RETURNED FROM DEBUG
READY
>_
```

CMD"R" (start clock [enable interrupts])

CMD"R"

Execute this command immediately after completion of a cassette input/output operation to re-start the real-time clock. See CMD"T".

CMD"S" (return to TRSDOS)

CMD"S"

Execute this command to initiate a normal return to the Operating System command mode. This will not re-initialize the system, but merely get you out of BASIC.

Be sure to save any BASIC program on disk or tape before using CMD"S", as your resident BASIC program will be lost.

CMD"T" (stop clock [disable interrupts])

CMD"T"

You must execute this command immediately before any BASIC tape input/output operation. Such operations are timing sensitive and cannot allow the interrupt-driven tasks (such as the real-time clock, TRACE, and CLOCK-display) to "steal" time.

Here are the commands which must be preceded by execution of CMD"T":

CLOAD	CLOAD?
INPUT#-1	CSAVE
INPUT#-2	PRINT#-1
SYSTEM	PRINT#-2

After completion of these operations, you can execute a CMD"R" to re-enable interrupts.

Example:

```
10 OPEN"I",1,"TEST/BAS"
20 CMD"T": INPUT#1,A,B,C
30 CMD"R"
```

Note: After CMD"D", you can use CMD"T" to prevent BASIC from transferring control to the DEBUG program when BREAK is pressed.

DEF FN (define function)

DEF FN *var1*(*var2*[,*var*...])=*exp*

where *var1* will be the name of the function, and is any valid LEVEL II variable name
var2 and subsequent var-items are used in defining what the function does
exp is an expression usually involving the variable(s) passed on the left of the equals sign

This statement lets you create your own implicit functions. That is, you only have to call it by name and the implicit function you defined will automatically be performed. Once a function has been defined with the DEF FN statement, you can call it simply by referencing the function name prefixed by FN. You can use it exactly as you'd use one of the intrinsic functions, e.g., SIN, ABS, STRING$.

The type of variable used to name the function determines what type of value the function will return. For example, if the function name has the single-precision attribute, then that function will return a single-precision value – regardless of the precision of the arguments.

Examples:

```
DEFFNA$(TITLE$,GRAPHICS%)=STRING$(LEN(TITLE$),GRAPHICS%)
```

The function A$ will require two arguments – one integer, one string; and it will return a string value.

```
DEFFNRC!(A)=1/(A*A)
```

The function RC! requires one argument, and returns a single-precision value, regardless of the precision of the argument.

The particular variable names you use as arguments in the DEF FN statement are not assigned to the function; when you call the function later, any valid variable name of the same type can be used. Furthermore, using a variable as an argument in a DEF FN statement has no effect on the value of that variable.

The function must be defined with at least one argument – even if this argument is not actually used to pass a value to the function. For example:

```
DEF FNR(A)=RND(0)
```

Examples:

```
10 DEFFNMLT(A,B)=A*B
20 INPUT "ENTER ARGUMENTS";X,Y
30 PRINT "PRODUCT IS"; FNMLT(X,Y)
```

Notice that FNMLT is defined with arguments A,B, but that when the function is called in line 30, variables X and Y are used. Any two valid variable names can be used to pass values to the function.

```
DEF FNRR(A,B)=A+INT(B*RND(0))
```
Returns a random number between A and B.

```
DEF FNL8$(A$)=LEFT$(A$,8)
```
Returns first 8 characters of string argument

```
DEF FNX#(A#,B#)=(A#-B#)*(A#-B#)
```
Returns double-precision value of "the square of the difference"

```
100 '          PROGRAM: STRING
110 ' EXAMPLE OF A STRING DEFFN FUNCTION
120 '
130 ' ******* FUNCTION TO CONCATENATE STRINGS *******
135 '
140 DEF FNADD$ (A$, B$) = A$ + " " + B$
150 CLS: PRINT TAB(15); "STRING DEFFN EXAMPLE"
160 PRINT: F$="": INPUT "ENTER FIRST NAME"; F$
165 IF F$="" THEN END
170 INPUT "ENTER LAST  NAME"; L$
180 '
190 ' ******* ADD F$ TO L$ WITH 1 BLANK IN BETWEEN *******
200 '
210 Z$ = FNADD$ (F$, L$)
220 PRINT TAB(6); "FULL NAME: "; Z$
230 GOTO 160
```

```
RUN ENTER

                STRING DEFFN EXAMPLE

ENTER FIRST NAME? JOHN ENTER
ENTER LAST  NAME? DOE ENTER
      FULL NAME: JOHN DOE
```

```
100 '          PROGRAM: MINMAX
110 ' EXAMPLE OF DEFFN FEATURE
120 '
130 ' ******* DEFINE MIN AND MAX FUNCTIONS *******
135 '
140 DEF FNMIN (A, B) = (A + B - ABS (A - B)) / 2
150 DEF FNMAX (A, B) = (A + B + ABS (A - B)) / 2
160 '
170 ' ******* READ 1ST VALUE - CALL IT THE MIN AND MAX *****
180 '          MN IS CURRENT MINIMUM VALUE
190 '          MX IS CURRENT MAXIMUM VALUE
200 '
210 READ MN: MX = MN
220 '
230 ' ******* GET NEXT VALUE AND FIND NEW MIN/MAX ********
240 '
250 READ V: IF V = 99999 THEN 320   'V=99999 MEANS ALL DONE
260 MN = FNMIN (MN, V)   'GET NEW MINIMUM
270 MX = FNMAX (MX, V)   'GET NEW MAXIMUM
280 GOTO 250
290 '
300 ' ******** PRINT RESULTS *******
310 '
320 PRINT "MINIMUM VALUE =", MN
330 PRINT "MAXIMUM VALUE =", MX
340 '
350 ' ******* DATA FOLLOWS - LAST VALUE MUST BE 99999 *******
360 '
370 DATA 1.2, 2, 3, 4.7, 5.332, 0.314, 6, 7, 8.3, 9.57, 99999
```

```
>RUN ENTER
MINIMUM VALUE =  .314
MAXIMUM VALUE =  9.57
READY
>370 DATA -1, 9, 0, 8, 1, 7, 2, 6, 3, 5, 4, 99999 ENTER
>RUN ENTER
MINIMUM VALUE = -1
MAXIMUM VALUE =  9
READY
>_
```

DEFUSR
(define entry address for USR routine)

DEFUSR*n*=*nmexp*

where *n* equals one of the digits 0,1,...,9; if *n* is omitted, 0 is assumed

nmexp specifies the entry address to a machine-language routine.

This statement lets you define the entry points for up to 10 machine-language routines. (In LEVEL II, where only one USR routine is available, the entry point address is POKEd into RAM.)

Example:

```
100 DEFUSR3=&H7D00
```

Assigns the entry point 7D00 hex, 32000 decimal, to the USR3 call. When your program calls USR3, control will branch to your sub-routine beginning at hex 7D00.

Here are three ways to get a machine language program into RAM so that it can be accessed via a USR*n* call:

1) Use the TRS-80 Editor Assembler, Radio Shack Catalog Number 26-2002, to convert the source code into an object file on tape; then load the tape under the SYSTEM command (use MEMORY SIZE to protect the code from destruction by BASIC).
2) Use the TRSDOS DEBUG program to type in the machine-code routine (then DUMP it to disk for safe-keeping); call DISK BASIC and answer MEMORY SIZE so as to protect the routine.
3) Have your DISK BASIC routine POKE the routine (decimal values for each byte) into high RAM. MEMORY SIZE should be set during initialization to protect the area you will POKE into.

See USR*n*.

INSTR (string search function)

INSTR([*n*,]*exp1$*,*exp2$*)

where *n* specifies a position in exp1$ where the search is to begin; if *n* is not supplied, 1 is assumed. (Position 1 is defined as the first character in the string.)
exp1$ is the string to be searched
exp2$ is the substring you want to search for

This function lets you search through a string to see if it contains another string. If it does, INSTR returns the starting position of the substring in the target string; otherwise zero is returned. Note that the entire substring must be contained in the search string, or zero is returned. Also note that INSTR only finds the first occurrence of a substring, starting at the position you specify.

Examples (let A$="ABCDEFG"):

Expression	Result
INSTR(A$,"BCD")	2
INSTR(A$,"12")	0
INSTR(A$,"ABCDEFGH")	0
INSTR(3,"1232123","12")	5

See the EDIT program under MID$= for a sample use of INSTR.

LINE INPUT (input a line from keyboard)

LINE INPUT[*"prompt"*] ;*var$*

where *prompt* is a prompting message

var$ is the name that will be assigned to the line you type in

LINE INPUT (or LINEINPUT – the space is optional) is similar to INPUT, except:

- When the statement is executed, and the Computer is waiting for keyboard input, no question mark is displayed
- Each LINE INPUT statement can assign a value to just one variable
- Commas and quotes will be accepted as part of the string input
- Leading blanks are not ignored – they become part of *var$*
- The only way to terminate the string input is to press **ENTER**

LINE INPUT is a convenient way to input string data without having to worry about accidental entry of delimiters – because only the **ENTER** key serves as a delimiter. If you want anyone to be able to input information to a program without special instructions, use LINE INPUT and then analyze the resultant string.

Some situations require that you input commas, quotes and leading blanks as part of the data. LINE INPUT serves well in such cases.

Examples:

```
LINE INPUT A$
```

Input A$ without displaying any prompt.

```
LINE INPUT"LAST NAME, FIRST NAME?";N$
```

Displays a prompt message and inputs data. Commas will not terminate the input string.

Try the following program to get the idea of LINE INPUT.

```
100 '          PROGRAM: LNINPUT
110 ' EXAMPLE OF LINEINPUT STATEMENT
120 '
130 CLEAR 300: CLS
140 PRINT TAB(15); "LINE INPUT STATEMENT": PRINT
150 PRINT: PRINT "*** ENTER TEXT ***"
151 '
152 ' *** GET STRING, THEN PRINT IT ***
153 '
155 A$=""    'SET A$ TO NULL STRING
```

```
160 LINEINPUT "==> "; A$
165 IF A$="" THEN END   'IF STILL NULL STRING, STOP!
170 PRINT A$
180 GOTO 155
```

```
RUN ENTER
```

```
                LINE INPUT STATEMENT

*** ENTER TEXT ***
==>                EXAMPLE TEXT
THIS TEXT HAS EMBEDDED LINE FEEDS AND TABS
IN      IT.  LINEINPUT ALSO ALLOWS DELIMITER (, :;"' ETC). ENTER
           EXAMPLE TEXT
THIS TEXT HAS EMBEDDED LINE FEEDS AND TABS
IN      IT.  LINEINPUT ALSO ALLOWS DELIMITER (, :;"' ETC).
==> ENTER
READY
>_
```

MID$= (replace portion of string)

MID$(*var$*,*n1*[,*n2*]) = *exp$*

where *var$* names the string to be changed
n1 specifies the starting position for the replacement
n2 specifies how many characters are to be replaced; if n2 is omitted, LEN(*exp$*) or LEN(*var$*)–*n1*+1 is used, whichever is smaller.

This statement lets you replace any part of a string with a specified substring, giving you a powerful string-editing capability.

Note that the length of the target string (*var$*) is never changed by the MID$= statement. If the replacement string, *exp$*, is too long to fit in the specified portion of var$, then the extra characters at the right of *exp$* are ignored.

However, if you specify the number of characters to be replaced, and this number is larger than the replacement string, then the length of the replacement string overrides the length you specified.

A$="ABCDEFG" at the beginning of each example below:

Ex. #	Expression	Resultant A$
1	MID$(A$,3,4)="12345"	AB1234G
2	MID$(A$,1,2)=""	ABCDEFG
3	MID$(A$,5)="12345"	ABCD123
4	MID$(A$,5)="01"	ABCD01G
5	MID$(A$,1,3)="***"	***DEFG

In example 2, the specified replacement length exceeds the length of the replacement string (which is zero); therefore the replacement-string length is used. In effect, no characters are replaced.

Sample program: EDIT

This program accepts an initial string, asks for a replacement position and a replacement string. Then it performs the MID$= replacement and prints the new string. Type in a position equal to zero to stop the program.

```
100 '          PROGRAM: EDIT
110 ' EXAMPLE OF INSTR FUNCTION FOR TEXT EDITTING
115 '
120 CLEAR 800: CLS
130 PRINT TAB(15); "  STRING-FUNCTION EDITOR"
135 '
140 ' ******* GET INITIAL TEXT *******
145 '
150 PRINT: PRINT "ENTER INITIAL TEXT STRING"
160 S$="": LINE INPUT S$: IF S$="" THEN END
165 '
170 ' ******* GET TARGET & REPLACEMENT STRINGS *******
175 '
180 T$="": PRINT: LINE INPUT"     TARGET STRING  "; T$
185 IF T$="" THEN END
190 LINE INPUT "REPLACEMENT STRING  "; R$
195 IF LEN(T$)<>LEN(R$)THEN PRINT"CAN'T CHANGE STRING LENGTH":
    GOTO 180
200 ' ******* MAKE REPLACEMENT(S) AND PRINT NEW STRING *****
210 I=1   'VARIABLE I POSITIONS TO BEGINNING POINT OF SEARCH
220 I=INSTR(I, S$, T$): IF I=0 THEN 150   'I=0 IF NOT FOUND
230 MID$(S$,I)=R$   'MAKE REPLACEMENT
240 PRINT "POSITION - "; I: PRINT S$
250 I=I+LEN(R$): GOTO 220   'ADVANCE POSITION
```

```
RUN ENTER

                STRING-FUNCTION EDITOR

ENTER INITIAL TEXT STRING
CHANGE "DISC" TO "DISK" EACH TIME IT OCCURS. . (DISC=>DISK)

     TARGET STRING  DISC ENTER
REPLACEMENT STRING  DISK ENTER
POSITION -  9
CHANGE "DISK" TO "DISK" EACH TIME IT OCCURS. . . (DISC=>DISK)
POSITION -  48
CHANGE "DISK" TO "DISK" EACH TIME IT OCCURS. . . (DISK=>DISK)

ENTER INITIAL TEXT STRING ENTER

READY
>_
```

TIME$ (get value of Real-Time Clock)

TIME$

TIME$ is a function with no arguments – when executed, it returns a string-value composed of the date and time currently stored in the Real-Time Clock memory area. The string is always 17 characters long and has the following format:

MM/DD/YY␣HH:MM:SS (month/day/year hr:min:sec)

The hour appears in 24-hour form, e.g., 1:30 PM appears as 13:30.

To set the time and date, get into the DOS READY mode and use the TRSDOS commands, TIME and DATE, as follows (assume it's 3:30 PM on January 1, 1979):

```
TIME 15:30:00 ENTER
DATE 01/01/79 ENTER
```

Or, you can set the time and date under DISK BASIC, by POKEing the time and date values into the appropriate addresses (see CLOCK, **TRSDOS Library Commands**).

TIME$ can be printed or used internally by your program in dedicated applications.

Examples:

```
1000 IF LEFT$(TIME$,15)="07/04/79 20:00"THEN 2000
1010 GOTO 1000
2000 REM...IT'S 8PM ON JULY 4TH, 1979
2010 REM...START FIREWORKS DISPLAY
.
.
.
```

The following program, CLOCK, will display the time and date until you press the @-key.

```
100 '          PROGRAM: CLOCK
110 ' EXAMPLE OF TIME$
120 '
130 CLS: PRINT CHR$ (23)   'GET INTO 32 CHARACTER MODE
140 '
150 ' ******* PRINT TIME AND DATE *******
160 '
170 PRINT @ 264, "THE TRS-80 TIME IS";
180 PRINT @ 458, "DATE: ";  LEFT$ (TIME$, 8);
190 PRINT @ 586, "TIME: "; RIGHT$ (TIME$, 8);
200 '
210 ' ******* STOP IF "@" KEY IS DEPRESSED *******
220 '
230 A$=INKEY$: IF A$ = "@" THEN END ELSE 180
```

USRn (call to user's external subroutine)

USR[*n*](*nmexp*)

where *n* specifies one of ten available USR calls, *n*=0,1,2,...,9. If *n* is omitted, zero is assumed.

nmexp is in the range < –32768 +32767> and is passed as an integer argument to the routine

These functions (USR0 through USR9) transfer control to machine-language routines previously defined with DEFUSR*n* statements.

When a USR call is encountered in a statement, control goes to the address specified in the DEFUSR*n* statement. This address specifies the entry point to your machine-language routine. A RET or JP 0A9A instruction in the routine returns control to the USR call in your BASIC program.

Note: If you call a USR*n* routine before defining the routine entry point with DEFUSR*n*, an ILLEGAL FUNCTION CALL error will occur.

You can pass one argument and retrieve one output value directly via the USR argument; or you can pass and retrieve arguments indirectly via POKE and PEEK statements.

Example:

```
10 DEFUSR1=&H7D00
20 REM...MORE PROGRAM LINES HERE
100 A=USR1(X)
```

The effect of this sequence is to:

1) Define USR1 as a routine with an entry point at hex 7D00 (line 10)
2) Transfer control to the routine; the value X can be passed to the routine if the routine makes the CALL described below (line 100)
3) When the routine returns to BASIC, the variable A may contain the value passed back from the routine (if your routine makes the JUMP described below); otherwise A will be assigned the value of X (line 100).

Passing arguments to and from USR routines

There are several ways to pass arguments back and forth between your BASIC main program and your USR routines: the two major ways are listed below.

1. POKE the argument(s) into fixed RAM locations. The machine-language routine can then access these values and place results in other RAM locations. When the routine returns control to BASIC, your program can PEEK into these addresses to pick up the "output" values. **This is the only way to pass two or more arguments back and forth.**

2. Pass one argument to the routine as the argument in the USRn call, then use special ROM calls to access this argument and return a value to BASIC. **This method is limited to sending one argument and returning one value** (both are integers).

ROM Calls

CALL OA7FH — Puts the USR argument into the HL register pair; H contains msb, L contains lsb. This CALL should be the first instruction in your USR routine.

JP 0A9AH — Use this JUMP to return to BASIC; the integer in HL becomes the output of the USR call. If you don't care about returning HL, then execute a simple RETurn instruction instead of this JUMP.

Examples:

Listed below is an assembled machine-language routine that will accept the argument from the USR call in BASIC, left-shift it one position, and return the result to BASIC.

```
                00100 ;
                00110 ; SHIFT FUNCTION
                00120 ;
                00130 ;       MACHINE CODE PROGRAM TO LEFT SHIFT
                00140 ;       AN ARGUMENT SENT FROM BASIC AND RETURN
                00150 ;       THE RESULT BACK TO BASIC
                00160 ;
7D00            00170         ORG     7D00H
                00180 ;
                00190 ; EQUATES AND ENTRY POINTS
                00200 ;
0A7F            00210 GETARG  EQU     0A7FH           ;GET ARGUMENT FROM BASIC
0A9A            00220 PUTANS  EQU     0A9AH           ;RETURN ANSWER TO BASIC
                00230 ;
7D00 CD7F0A     00240 SHIFT   CALL    GETARG          ;GET NUMBER FROM BASIC
7D03 CB15       00250         RL      L               ;SHIFT L
7D05 CB14       00260         RL      H               ;SHIFT H - ANSWER IN HL
7D07 C39A0A     00270         JP      PUTANS          ;RETURN TO BASIC W/ ANSWER
                00280 ;
7D00            00290         END     SHIFT
```

The following program includes the decimal code for the SHIFT routine. The code is POKEd into RAM and then accessed as a USR routine. RUN the program; to stop, enter a value of zero.

Note: The following two BASIC programs require that you reserve memory addresses above 31999 for the USR Code. (Answer MEMORY SIZE? with 31999.)

```
100 '          PROGRAM: SHIFT
110 ' MACHINE LANGUAGE USER FUCTION TO LEFT SHIFT
120 '
130 ' ******* MACHINE CODE AT 7D00 HEX *******
140 '
150 DEFUSR5 = &H7D00
160 '
170 ' ******* POKE USER PROGRAM INTO MEMORY *******
180 '
190 FOR X = 32000 TO 32009    '7D00 HEX EQUALS 32000 DECIMAL
200     READ A
210     POKE X,A
220 NEXT X
230 '
240 ' ******* GET VALUE FROM USER *******
250 '
260 CLS: PRINT TAB(15); "USR5 LEFT-SHIFT FUNCTION"
270 PRINT: INPUT"ENTER INTEGER VALUE"; V
280 IF V=0 THEN END
290 PRINT "LEFT SHIFTED VALUE = "; TAB(32); USR5(V)
300 GOTO 270
310 '
320 ' ******* DATA IS DEMICAL CODE FOR HEX PROGRAM *******
330 '
340 DATA 205,127,10,203,21,203,20,195,154,10
```

```
RUN ENTER
```

```
                USR5 LEFT-SHIFT FUNCTION

ENTER INTEGER VALUE? 7 ENTER
LEFT SHIFTED VALUE =              14

ENTER INTEGER VALUE? -7 ENTER
LEFT SHIFTED VALUE =             -13

ENTER INTEGER VALUE? 32767 ENTER
LEFT SHIFTED VALUE =              -2

ENTER INTEGER VALUE? 0 ENTER
READY
>_
```

Listed below is an assembled program to white out the display (an "inverse" CLEAR key!).

```
                00100 ;
                00110 ; ZAP OUT SCREEN USR FUNCTION
                00120 ;
7D00            00130         ORG     7D00H
                00140 ;
                00150 ; EQUATES
                00160 ;
3C00            00170 VIDEO   EQU     3C00H           ;START OF VIDEO RAM
00BF            00180 WHITE   EQU     0BFH            ;ALL WHITE GRAPHICS BYTE
03FF            00190 COUNT   EQU     3FFH            ;NUMBER OF BYTES TO MOVE
                00200 ;
                00210 ; PROGRAM CHAIN MOVES X'BF' INTO ALL OF VIDEO RAM
                00220 ;
7D00 21003C     00230 ZAP     LD      HL,VIDEO        ;SOURE ADDRESS
7D03 36BF       00240         LD      (HL),WHITE      ;PUT OUT 1ST BYTE
7D05 11013C     00250         LD      DE,VIDEO+1      ;DESTINATION ADDRESS
7D08 01FF03     00260         LD      BC,COUNT        ;NUMBER OF ITERATIONS
7D0B EDB0       00270         LDIR                    ;DO IT TO IT!!!
                00280 ;
7D0D C9         00290         RET                     ;RETURN TO BASIC
7D00            00300         END     ZAP
```

This routine can be POKEd into RAM and accessed as a USR routine, as follows.

```
100 '          PROGRAM: USR1
110 ' EXAMPLE OF A USER MACHINE LANGUAGE FUNCTION
115 ' DEPRESS THE '@' KEY WHILE NUMBERS ARE PRINTING TO STOP
120 '
130 ' ******* POKE MACHINE PROGRAM INTO MEMORY *******
140 '
150 DEFUSR1 = &H7D00
160 FOR X = 32000 TO 32013   '7D00 HEX EQUAL 32000 DECIMAL
170     READ A
180     POKE X, A
190 NEXT X
192 '
194 ' ******* CLEAR SCREEN & PRINT NUMBERS 1 THRU 100 *******
196 '
200 CLS
205 PRINT TAB(15); "WHITE-OUT USER ROUTINE": PRINT
210 FOR X = 1 TO 100
220     PRINT X;
225     A$ = INKEY$: IF A$ = "@" THEN END
230 NEXT X
240 '
250 ' ******* JUMP TO WHITE-OUT SUBROUTINE *******
260 '
270 X = USR1 (0)
280 FOR X = 1 TO 1000: NEXT X   'DELAY LOOP
290 GOTO 200
300 '
310 ' ******* DATA IS DEMICAL CODE FOR HEX PROGRAM *******
320 '
330 DATA 33,0,60,54,255,17,1,60,1,255,3,237,176,201
```

RUN the program. An equivalent BASIC white out routine takes a long time by comparison!

Disk-Related Features

Programs and data are stored as "files" under TRSDOS. Each program or data-set on the disk has its own, distinct file specification – which includes a name plus identifying information.

Before attempting any disk input/output – including loading and saving BASIC programs, refer to the **TRSDOS Overview**. Also, review the Notation Conventions described under **General Information**. That's the only way to be sure you understand the statement syntax descriptions.

DISK BASIC provides a powerful set of commands, statements and functions relating to disk I/O under TRSDOS. These fall into two categories:

1. File manipulation: dealing with files as units, rather than with the distinct records the files contain.

2. File access: preparing data files for I/O; reading and writing to the files.

Commands discussed under "File Manipulation":

KILL	delete a program or data file from the disk
LOAD	load a BASIC program from disk
MERGE	merge an ASCII-format BASIC program on disk with one currently in RAM
RUN"*program*"	load and execute a BASIC program stored on disk
SAVE	save the resident BASIC program on disk

Statement and functions discussed under "File Access":

Statements

OPEN	Open a file for access (create the file if necessary)
CLOSE	Close access to the file
INPUT #	Read from disk, sequential mode
LINE INPUT#	Read a line of data, sequential mode
PRINT#	Write to disk, sequential mode
GET	Read from disk, random access mode
PUT	Write to disk, random access mode
FIELD	Assign field sizes and names to random access file buffer
LSET	Place value in specified buffer field, add blanks on the right to fill field
RSET	Place value in specified buffer field, add blanks on the left to fill field

Functions

CVD	Restore double-precision number to numeric form after GETting from disk
CVI	Restore integer to numeric form after GETting from disk
CVS	Restore single-precision number to numeric form after GETting from disk
EOF	Check to see if end of file encountered during read
LOF	Return number of last record in file
MKD$	Convert double-precision number to string so it can be PUT on disk
MKI$	Convert integer to string so it can be PUT on disk
MKS$	Convert single-precision number to string so it can be PUT on disk

File Manipulation

KILL (delete a file from the disk)

KILL *exp$*

where *epx$* defines a file specification for an existing file

This command works like the TRSDOS KILL command – see **TRSDOS Library Commands.**

Example:
```
KILL"OLDFILE/BAS. PSW1
```
deletes the file specified from the first drive which contains it.

Do not KILL an open file, or you may destroy the contents of the diskette. (First CLOSE the open file.)

LOAD (load BASIC program file from disk)

LOAD *exp$* [,R]

where *exp$* defines a filespec for a BASIC program file stored on disk

R tells BASIC to RUN the program after it is loaded

This command loads a BASIC program file into RAM; if the R option is used, BASIC will proceed to RUN the program automatically; otherwise, BASIC will return to the command mode.

LOAD without the R option wipes out any resident BASIC program, clears all variables, and closes all open files. LOAD with the R option deletes the resident program and clears all variables, but does not close the open files.

LOAD with the R option is equivalent to the command RUN *exp$*,R. Either of these commands can be used inside programs to allow program chaining – one program calling another, etc.

If you attempt to LOAD a non-BASIC file, a DIRECT STATEMENT IN FILE or LOAD FORMAT ERROR will occur.

Examples:

LOAD"PROG1/BAS:2"	Clears resident BASIC program and loads PROG1/BAS from drive 2; returns to BASIC command mode.
10 REM... INSTRUCTIONS	Example of chaining two programs – the first may be used to give instructions and then to load the "working" part of the program (PROG2/BAS). Note that line 1000 is equivalent to: 1000 RUN"PROG2/BAS"
1000 LOAD"PROG2/BAS",R	

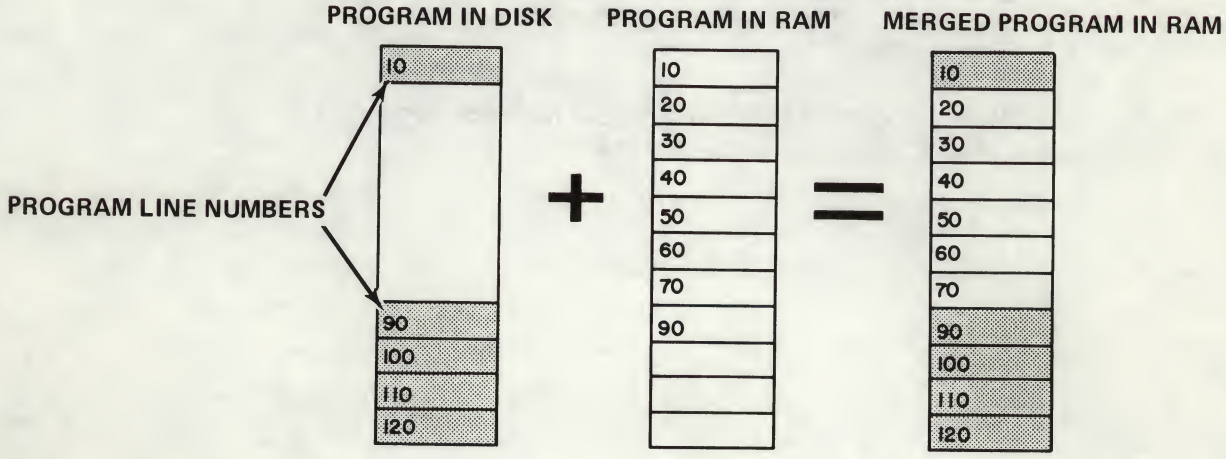
MERGE
(merge disk program with resident program)

MERGE *exp$*

where *exp$* defines a filespec for an ASCII-format BASIC disk file, e.g., a program saved with the A-option.

MERGE is similar to LOAD – except that the resident program is not wiped out before the new program *exp$* is loaded. Instead, *exp$* is merged into the resident program.

That is, program lines in *exp$* will simply be inserted into the resident program in sequential order. If line numbers in *exp$* coincide with line numbers in the resident program, the resident lines will be replaced by those from *exp$*.



MERGE provides a convenient means of putting modular programs together. For example, an often-used set of BASIC subroutines can be tacked onto a variety of programs with this command.

For example, suppose the following program is in RAM:

```
10 REM...MAIN PROGRAM
20 GOSUB 1000
30 REM...MORE PROGRAM LINES HERE
999 END
1000 REM...NEED TO ADD SUBROUTINES HERE
1010 REM...SO USE MERGE COMMAND
1020 PRINT"SUBROUTINE NOT AVAILABLE":RETURN
```

And suppose the following program is stored on disk in ASCII format:

```
1000 REM...BEGINNING OF SUBROUTINE
1010 PRINT"EXECUTING SUBROUTINE..."
1020 REM...MORE PROGRAM LINES HERE
1100 RETURN
```

Assuming the subroutine program is named SUB/TXT, then we could MERGE it with the statement:

MERGE"SUB/TXT"

and the resultant program in RAM would be:

```
10 REM...MAIN PROGRAM
20 GOSUB 1000
30 REM...MORE PROGRAM LINES HERE
999 END
1000 REM...BEGINNING OF SUBROUTINE
1010 PRINT"EXECUTING SUBROUTINE..."
1020 REM...MORE PROGRAM LINES HERE
1100 RETURN
```

Note that MERGE closes all files and clears all variables. Upon completion, BASIC returns to the command mode.

RUN"program"
(load and execute a program from disk)

RUN *exp$* [,R]

where *exp$* defines the filespec for a BASIC program stored on disk. R leaves open files open

If the R-option is not selected, all open files will be closed.

When the command is executed, any resident BASIC program will be replaced by the program contained in *exp$*.

Example:

```
RUN"DISKDUMP/BAS" ENTER
```

Loads and executes the BASIC sector-dump program.

Suppose you save the following program on disk with the name "PROG1/BAS":

```
10 PRINT"PROG1 EXECUTING..."
20 RUN"PROG2/BAS"
```

And save this program on disk with the name "PROG2/BAS":

```
10 PRINT"PROG2 EXECUTING..."
20 RUN"PROG1/BAS"
```

Now type:
`RUN"PROG1/BAS ENTER`
and you'll see a simple example of program chaining.
Hold down the BREAK key to interrupt the program chain.

SAVE (save program onto disk)

SAVE *exp$* [,A]

where *exp$* defines the file-name and optional extension, password, and drive to be used. If the file-name already exists, its previous contents will be lost as the file is re-created.

A causes the file to be stored in ASCII rather than compressed-format.

This command lets you save your BASIC programs on disk. You can save the program in compressed or ASCII format.

Using compressed-format takes up less disk space and is faster during both SAVEs and LOADs. This is the way BASIC programs are stored in RAM.

Using the ASCII option makes it possible to do certain things that cannot be done with compressed-format BASIC files.

Examples:

- The MERGE command requires that the disk file be in ASCII form.
- You can use the TRSDOS commands LIST and PRINT with ASCII-format files.
- Programs which read in other programs as data will typically require that the data programs be stored in ASCII.

Useful conventions for placing extensions on BASIC programs:
For compressed-format programs, use the extension /BAS.
For ASCII format programs, use the extension /TXT.

Examples of SAVE command:

```
SAVE"FILE1/BAS.JOHNQDOE:3"
```
saves the resident BASIC program in compressed-format with the file name FILE1, extension /BAS, password .JOHNQDOE; the file is placed on drive :3.

```
SAVE"MATHPAK/TXT",A
```
saves the resident program in ASCII form, using the name MATHPAK/TXT, on the first non write-protected drive.

Upon completion of a SAVE, BASIC returns in the command mode.

File Access

This section is divided into four parts:

1) Creating files and assigning buffers – OPEN and CLOSE
2) Statements and functions
3) Sequential I/O techniques
4) Random I/O techniques.

If this is your first experience with disk file access, you should concentrate on parts 1, 3 and 4, perhaps just skimming through part 2 to get a general idea of how the functions and statements work. Later you can go back to part 2 and learn the details of statement and function syntax.

Creating files and assigning buffers

During the initialization dialog, you type in a number in response to HOW MANY FILES? The number you type in tells BASIC how many buffers to create to handle your disk accesses (reads and writes).

Each buffer is given a number from 1 to 15. If you type:

```
HOW MANY FILES? 4 ENTER
```

then BASIC sets aside four buffers, numbered 1,2,3 and 4.

You can think of a buffer as a waiting area that data must pass through on the way to and from the disk file. When you want to access a particular file, you must tell BASIC which buffer to use in accessing that file. You must also tell BASIC what kind of access you want – sequential output, sequential input, or random input/output.

All this is done with the OPEN statement, and "undone" with the CLOSE statement.

OPEN
(Assign a buffer to a file and set mode)

OPEN *exp1$,nmexp,exp2$*

where *exp1$* is a string expression or constant of which only the first character is significant; this character specifies the mode in which the file is to be opened:

exp1$=	access mode
I	sequential input
O	sequential output
R	random I/O

nmexp has a value from 1 to 15, and tells BASIC which buffer to assign to the file specified by *exp2$*

exp2$ defines a TRSDOS file specification

This statement makes it possible to access a file. *exp1$* determines what kind of access you'll have via the specified buffer; *nmexp* determines which buffer will be assigned to the file; and *exp2$* names the file to be accessed. If *exp2$* does not exist, then TRSDOS may or may not create it, depending on the access mode.

Note: *nmexp* (buffer number) cannot exceed the number you entered for the FILES? question during initialization. If you entered:

```
HOW MANY FILES? 2 ENTER
```

then *nmexp* can have the value 1 or 2.

Examples of OPEN statements:

```
100 OPEN "O",1,"CLIENTLS/TXT"
```

Opens the file "CLIENTLS/TXT" for sequential output. Buffer 1 will be used. If the file does not exist, it will be created. If it already exists, then its previous contents are lost. (This is explained under "Sequential I/O Techniques".)

```
100 OPEN "I",1,"PROG1/TXT:1"
```

Opens the file "PROG1/TXT" on drive 1 for sequential input. Buffer 2 is assigned to the file. If PROG1/TXT does not exist on drive 1, an error message is returned – since you can't input from a non-existent file!

```
100 INPUT"MODE (I,O,R)";MODE$
110 INPUT"BUFFER NUMBER";BUFFER%
120 INPUT"FILE SPECIFICATION";FILESPEC$
130 OPEN MODE$,BUFFER%,FILESPEC$
```

This sequence of statements lets you provide the arguments for the OPEN statement during program execution. The first character of MODE$ sets the access mode, BUFFER% determines which buffer will be used, and FILESPEC$ gives the file specification.

```
OPEN"R",2,"DATA/BAS.SPECIAL"
```

Opens the file DATA/BAS with password SPECIAL, in the random I/O mode, using buffer number 2. If DATA/BAS does not exist, it will be created on the first non write-protected drive.

While a file is open, it is referenced by the buffer-number which was assigned to it. Examples:

GET *buffer-number*
PUT *buffer-number*
PRINT#*buffer-number*
INPUT#*buffer-number*

All these statements will reference the file which was OPENed via *buffer-number*. **The mode must be correct.**

Once a buffer has been assigned to a file with the OPEN statement, that buffer cannot be used in another OPEN statement. You have to CLOSE it first.

More on Buffer Assignments

Two or more buffers may be assigned to the same file for sequential input (I-mode). However, only one buffer at a time may be assigned to a file for sequential output (O-mode) or random access R-mode.

For example:

```
10 OPEN "I",1,"TEST/TXT:1"
20 OPEN "I",2,"TEST/TXT:1"
```

Now TEST/TXT can be accessed via buffers 1 and 2 for sequential input.

CLOSE (close access to the file)

CLOSE [*nmexp* [,*nmexp* . . .]]

where *nmexp* has a value from 1 to 15, and refers to the file's buffer-number (assigned when the file was opened). If *nmexp* is omitted, all open files will be closed.

This command terminates access to a file through the specified buffer(s). If *nmexp* has not been assigned in a previous OPEN statement, then

CLOSE *nmexp*

has no effect.

Examples of CLOSE statements:

```
CLOSE 1,2,8
```

Terminates the file assignments to buffers 1, 2 and 8. These buffers can now be assigned to other files with OPEN statements.

```
CLOSE FIRST%+COUNT%
```

Terminates the file assignment to the buffer specified by the sum (FIRST% + COUNT%).

Do not remove a diskette which contains an open file – first close the file. This is because the last 256 bytes of data may not have been written to disk yet. Closing the file will write the data, if it hasn't already been written.

The following actions and conditions cause all files to be automatically closed:

NEW ENTER
RUN ENTER
MERGE *filespec* ENTER
EDITing a file
Adding or deleting program lines
Execution of the CLEAR *n* statement
Disk Errors

INPUT# (sequential read from disk)

INPUT# *nmexp*, *var*[,*var*...]

where *nmexp* specifies a sequential input file buffer, *nmexp*=1,2,...,15

var is the variable name to contain the data from the file

This statement inputs data from a disk file. The data is input sequentially. That is, when the file is first opened, a pointer is set to the beginning of the file. Each time data is input, the pointer advances. To start over reading from the beginning of the file, you must close the file-buffer and re-open it.

INPUT# doesn't care how the data was placed on the disk – whether a single PRINT# statement put it there, or whether it required 10 different PRINT# statements. What matters to INPUT# are the positions of the terminating characters and the EOF marker.

To INPUT# data successfully from disk, you need to know ahead of time what the format of the data is. Here is a description of how INPUT# interprets the various characters it encounters when reading data.

When inputting data into a variable, BASIC ignores leading blanks; when the first non-blank character is encountered, BASIC assumes it has encountered the beginning of the data item.

The data item ends when a terminating character is encountered or when a terminating condition occurs. The particular terminating characters vary, depending on whether BASIC is inputting to a numeric or string variable.

Special Note

Here's an important exception to keep in mind in reading the following material.

When <EN> (a carriage return) is preceded by <LF> (a line feed), the <EN> is not taken as a terminator. Instead, it becomes a part of the data item (string variable) or is simply ignored (numeric variable).

(To enter the <LF> character from the keyboard, press the down arrow. To enter the <EN> character, press ENTER.)

This exception applies to all cases noted below where <EN> is said to be a terminator.

Numeric Input

Suppose the data image on disk is

␣1.234␣-33␣␣27␣ <EN>

<EN> denotes a carriage-return character (ASCII code decimal 13).

Then the statement

```
INPUT#1, A,B,C
```

or the sequence of statements

```
INPUT#1,A: INPUT#1,B: INPUT#1,C
```

will assign the values as follows:

A=1.2345
B=–33
C=27

This works because blanks and <EN> serve as terminators for input to numeric variables. The blank before 1.2345 is a "leading blank", therefore it is ignored. The blank after 1.2345 is a terminator; therefore BASIC starts inputting the second variable at the – character, inputs the number –33, and takes the next two blanks as terminators. The third input begins at the 2 and ends with the 7.

String Input

When reading data into a string variable, INPUT ignores all leading blanks; the first non-blank character is taken as the beginning of the data item.

If this first character is a double-quote ("), then INPUT will evaluate the data as a quoted string: it will read in all subsequent characters up to the next double-quote. Commas, blanks, and <EN> –characters will be included in the string. The quotes themselves do not become a part of the string.

If the first character of the string item is not a double-quote, then INPUT will evaluate the data as an unquoted string: It will read in all subsequent characters up to the first comma, or <EN>. If double quotes are encountered, they will be included in the string.

For example, if the data on disk is:

PECOS,␣TEXAS "GOOD MELONS"

Then the statement

```
INPUT#1, A$, B$, C$
```

would assign values as follows:

A$=PECOS
B$=␣TEXAS "GOOD␣MELONS"
C$= null string

If a comma is inserted in the data image before the first double quote, C$ will get the value, GOOD MELONS.

These are very simple examples just to give you an idea of how INPUT works. However, there are many other ways to input data – different terminators, different target variable types, etc.

Rather than taking a shotgun approach and trying to cover them all, we'll give a generalized description of how input works and what the terminating characters and conditions are, and then provide several examples.

When BASIC encounters a terminating character, it scans ahead to see how many more terminating characters it can include with the first terminator. This ensures that BASIC will begin looking for the next data item at the correct place.

The list below defines the various terminating sets INPUT# will look for. It will always try to take-in **the largest set possible.**

Numeric-input terminator sets

end of file encountered
255th data character encountered
, (comma)
<EN>
<EN> <LF>
␣[␣...][<EN>]
␣[␣...][<EN><LF>]

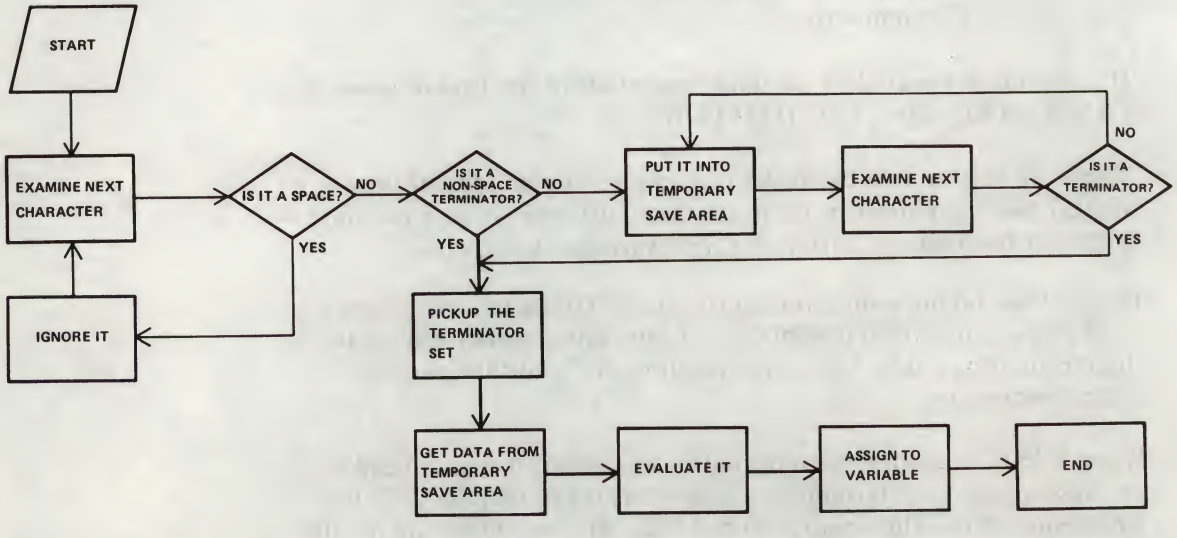
Quoted-string terminator sets

end of file encountered
255th data character encountered
" (double quote)
" [␣...][,]
" [␣...][<EN>]
" [␣...][<EN> <LF>]

Unquoted-string terminator sets

end of file encountered
255th data character encountered
,
<EN> [<LF>]

Here's a flow chart describing how INPUT# assigns data to a variable:



The following table shows how various data images will be read-in by the statement:

```
INPUT#1, A, B, C
```

Ex.#	Image on disk	Values assigned
1	␣123.45␣ <EN><LF>␣8.2E4␣␣7000<EN>	A=123.45 B=82000 C=7000
2	␣␣␣3<LF><EN> 4 <EN>5 <EN> A12eof	A=34 B=5 C=0
3	1,,2,3,4 <EN>	A=1 B=0 C=2
4	1,3,end-of-file	A=1 B=2 C=0 end of file error

In Example 2 above, why does variable C get the value 0? When the input reaches the end of file, it terminates the last data item, which then contains "A12". This is evaluated by a routine just like the BASIC VAL function –which returns a zero since the first character of "A12" is non-numeric.

In Example 3, when INPUT# goes looking for the second data item, it immediately encounters a terminator (the comma); therefore variable B is given the value zero.

The following table shows how various data images on disk will be read by the statement:

```
INPUT#1, A$, B$
```

Ex.#	Image on disk	Values assigned
1	␣␣␣"ROBERTS,J."ROBERTS,M.N eof	A$=ROBERTS,J. B$=ROBERTS,M.N.
2	␣␣␣ROBERTS,J.,␣␣␣ROBERTS,M.N. <EN>	A$=ROBERTS B$=J.
3	THE WORD "QUO",12345.789 <EN>	A$=THE WORD "QUO" B$=12345.789
4	BYTE<LF> <EN> UNIT OF MEMORY eof	A$=BYTE<LF> <EN> UNIT OF MEMORY B$=null (eof error)

In example 3, the first data item is an unquoted string, therefore the double-quotes are not terminators, and become part of A$.

In example 4, the <EN> is preceded by an <LF>, therefore it does not terminate the first string; both <LF> and <EN> are included in A$.

Technical Note: The above discussion ignores the role of the input buffer in the sequential input process. Actually, DISK BASIC always reads in 256-byte data records into the buffer, and then sorts through what's in the buffer to "satisfy" the INPUT# variable list. That's why

```
100 INPUT#1, A%
200 INPUT#1, B%
```

do not necessarily require two disk accesses. The 256-byte record in the buffer can contain enough data for A%, B% and more.

LINE INPUT#
(read a line of text from disk)

LINE INPUT#*nmexp*,*var$*

where *nmexp* specifies a sequential output file buffer, *nmexp*=1,2,...,15

var$ is the variable name to contain the string data

Similar to LINE INPUT from keyboard, this statement reads a "line" of string data into var$. This is useful when you want to read an ASCII-format BASIC program file as data, or when you want to read in data without following the usual restrictions regarding leading characters and terminators.

LINE INPUT (or LINEINPUT – the space is optional) reads everything from the first character up to:

1) an <EN> character which is not preceded by <LF>
2) the end-of-file
3) the 255th data character (this 255 character is included in the string)

Other characters encountered – quotes, commas, leading blanks, <LF> <EN> pairs – are included in the string.

For example, if the data looks like:

```
10 CLEAR 500   <EN>
20 OPEN "I",1,"PROG"   <EN>
.
.
.
```

then the statement

```
LINEINPUT#1,A$
```

could be used repetitively to read each program line, one line at a time.

PRINT# (sequential write to disk file)

PRINT#*nmexp*,[USING *format$*;] *exp*[*p exp* . . .]

where *nmexp* specifies a sequential output file buffer, *nmexp*=1,2,. . . ,15

format$ is a sequence of field specifiers used with the USING option

p is a delimiter placed between every two expressions to be PRINTed to disk; either a semi-colon or comma can be used (semi-colon is preferable)

exp is the expression to be evaluated and written to disk

This statement writes data sequentially to the specified file. When you first open a file for sequential output, a pointer is set to the beginning of the file, therefore your first PRINT# places data at the beginning of the file. At the end of each PRINT# operation, the pointer advances, so the values are written in sequence.

A PRINT# statement creates a disk image similar to what a PRINT to display creates on the screen. Remember this, and you'll be able to set up your PRINT# list correctly for access by one or more INPUT statements.

PRINT# does not compress the data before writing it to disk; it writes an ASCII-coded image of the data.

For example, if A=123.45

```
PRINT#1,A
```

will write a nine-byte character sequence onto disk:

␣123.45␣ <EN>

The punctuation in the PRINT list is very important. Unquoted commas and semi-colons have the same effect as they do in regular PRINT to display statements.

For example, if A=2300 and B=1.303, then

```
PRINT#1,A,B
```

places the data on disk as

␣2300␣␣␣␣␣␣␣␣␣␣1.303␣ <EN>

The comma between A and B in the PRINT# list causes 10 extra spaces in the disk file. Generally you wouldn't want to use up disk space this way, so you should use semi-colons instead of commas.

```
PRINT#1,A;B
```

writes the data as:

2300 1.303 <EN>

PRINT# with numeric data is quite straightforward – just remember to separate the items with semi-colons.

PRINT# with string data requires more care, primarily because you have to insert delimiters so the data can be read back correctly. In particular, you must separate string items with explicit delimiters if you want to INPUT# them as distinct strings.

For example, suppose:

A$="JOHN Q. DOE" and B$="100-01-001"

Then:

```
PRINT#1, A$;B$
```

would produce this image on disk:

JOHN Q. DOE100-01-001 <EN>

which could not be INPUT back into two variables.

The statement:

```
PRINT#1, A$;",";B$
```

would produce:

JOHN Q. DOE, 100-01-001

which could be INPUT# back into two variables.

This method is adequate if the string data contains no delimiters – commas or <EN> –characters. But if the data does contain delimiters or leading blanks that you don't want to ignore, then you must supply explicit quotes to be written along with the data. For example, suppose A$="DOE, JOHN Q." B$="100-01-001"

If you use

```
PRINT#1,A$;",";B$
```

the disk image will be:

DOE, JOHN Q.,100-01-001 <EN>

When you try to input this with a statement like

```
INPUT#2,A$,B$
```

A$ will get the value "DOE", and B$ will get "JOHN Q." – because of the comma after DOE in the disk image.

To write this data so that it can be input correctly, you must use the CHR$ function to insert explicit double quotes into the disk image. Since 34 is the decimal ASCII code for double quotes, use CHR$(34) as follows:

```
PRINT#1,CHR$(34);A$;CHR$(34);B$
```

this produces the disk image

"DOE, JOHN Q."100-01-001 <EN>

which can be read with a simple

```
INPUT#2,A$,B$
```

Note: You can also use the CHR$ function to insert other delimiters and control codes into the file, for example:

CHR$(10)	<LF> Line Feed
CHR$(13)	carriage return (<EN>character)
CHR$(11) or CHR$(12)	line-printer top-of-form

USING Option

This option makes it easy to write files in a carefully controlled format. You could create a report file this way, which then could be LISTed or PRINTed (TRSDOS commands).

Or you could use this option to control how many characters of a value are written to disk.

For example, suppose:

```
A$="LUDWIG"
B$="VAN"
C$="BEETHOVEN"
```

Then the statement

```
PRINT#1,USING"!. !.%  %";A$;B$;C$
```

would write the data in nickname form:

```
L.V.BEET    <EN>
```

(In this case, we didn't want to add any explicit delimiters.) See the PRINT USING description in the LEVEL II BASIC Reference Manual for a complete explanation of the field-specifiers.

Technical Note: The above discussion ignores the role of the output buffer in the sequential write process. Actually, the data is first placed into the buffer, and then, as 256-byte records are filled, the data is written to the disk file. That's why there isn't always a disk access during execution of each PRINT# statement.

Random Access Statements

FIELD
(organize a random file-buffer into fields)

FIELD *nmexp,nmexp1* AS *var1$* [,*nmexp2* AS *var2$* . . .]

where *nmexp*	specifies a random access file buffer, *nmexp*=1,2,. . . ,15
nmexp1	specifies the length of the first field,
var1$	defines a variable name for the first field
nmexp2	specifies the length of the second field
var2$	defines a variable name for the second field
. . .	subsequent *nmexp* AS *var$* pairs define other fields in the buffer

Before FIELDing a buffer, you must use an OPEN statement to assign that buffer to a particular disk file (must use random access mode). Then use the FIELD statement to organize a random file buffer so that you can pass data from BASIC to disk storage and vice-versa.

Each random file buffer has 255 bytes which can store data for transfer from disk storage to BASIC or from BASIC to disk. However, you need a way to access this buffer from BASIC so that you can either read the data it contains or place new data in it. The FIELD statement provides the means of access.

You may use the FIELD statement any number of times to "re-organize" a file buffer. FIELDing a buffer does not clear the contents of the buffer; only the means of accessing the buffer (the field names) are changed. Furthermore, two or more field names can reference the same area of the buffer.

Examples:

```
FIELD 1, 255 AS A$
```

This statement tells BASIC to assign the entire 255-byte buffer to the string variable A$. If you now print A$, you will see the contents of the buffer. Of course, this value would be meaningless unless you have used GET to read a 255-byte record from disk.

Note: All data – both strings and numbers – must be placed into the buffer in string form. There are three pairs of functions (MKI$/CVI,MKS$/CVS,MKD$/CVD) for converting numbers to strings and vice-versa. See "Functions", below.

```
FIELD 3, 16 AS NM$, 25 AS AD$, 10 AS CY$, 2 AS ST$,7 AS ZP$
```

The first 16 bytes of buffer 3 are assigned the buffer name NM$; the next 25, AD$; the next 10, CY$; the next 2, ST$; and the next 7, ZP$. The remaining 195 bytes of the buffer are not fielded at all.

More on field names

Field names, like NM$,AD$,CY$,ST$ and ZP$, are not string variables in the ordinary sense. They do not consume the string space available to BASIC.

Instead, they point to the buffer field which you assigned with the FIELD statement. That's why you can use:

```
100 FIELD 1,255 AS A$
```

without worrying about whether 255 bytes of string space are available for A$.

If you use a buffer field name on the left side of an ordinary assignment statement, that name will no longer point to the buffer field; therefore you won't be able to access that field using the previous field name.

For example,

```
A$=B$
```

nullifies the effect of the FIELD statement above (line 100).

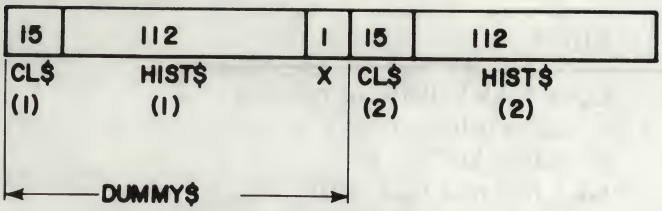
During random input, the GET statement places data into the 255-byte buffer, where it can be accessed using the field names assigned to that buffer. During random output, LSET and RSET place data into the buffer, so you can then PUT the buffer contents into a disk file.

Often you'll want to use a dummy variable in a FIELD statement to "pass over" a portion of the buffer and start fielding it somewhere in the middle. For example:

```
FIELD 1,15 AS CLIENT$(1),112 AS HIST$(1)
FIELD 1,128 AS DUMMY$,15 AS CLIENT$(2),112 AS HIST$(2)
```

In the second FIELD statement, DUMMY$ serves to move the starting position of CLIENT$(2) to position 129. In this manner, two identical "subrecords" are defined on buffer number 1. We won't actually use DUMMY$ to place data into the buffer or retrieve it from the buffer.

The buffer now "looks" like this:



Note that only one byte (the 128th byte) is left unused in this field structure.

GET
(read a record from disk – random access)

GET *nmexp1*[,*nmexp2*]

where *nmmexp1* specifies a random access file buffer, *nmexp1*=1,2,. . .,15

nmexp2 specifies which record to GET in the file; if omitted, the current record will be read.

This statement gets a data record from a disk file and places it in the specified buffer. Before GETting data from a file, you must open the file and assign a buffer to it. That is, a statement like:

OPEN "R",*nmexp1*,*filespec*

is required **before** the statement:

GET *nmexp1*,*nmexp2*

When BASIC encounters the GET statement, it looks at the buffer's control block, and obtains:

- the information needed to access the file
- the mode in which this buffer was set up (must be R)
- the current record number
- The EOF (end-of-file) record number, i.e., the highest numbered record in the file
- lots of other information for internal use

BASIC then reads record *nmexp2* from the file and places it into the buffer. If you omit the record number, it will read the current record.

The "current record" is the record whose number is one higher than that of the last record accessed. The first time you access a file via a particular buffer, the current record is set equal to 1.

For example:

Program statement	Effect
1000 OPEN"R",1,"NAME/BAS"	Open NAME/BAS for random access using buffer 1
1010 FIELD 1,...	Structure buffer
1020 GET 1	GET record 1 into buffer 1
1025 REM ... ACCESS BUFFER	
1030 GET 1,30	GET record 30 into buffer 1
1035 REM ... ACCESS BUFFER	
1040 GET 1,25	GET record 25 into buffer 1
1046 REM ... ACCESS BUFFER	
1050 GET 1	GET record 26 into buffer 1

If you attempt to GET a record whose number is higher than that of the end-of-file record, BASIC will fill the buffer with hex zeroes, and no error will occur.

To prevent this from occurring, you can use the LOF function to determine the number of the highest numbered record.

PUT
(write a record to disk – random access)

PUT *nmexp1*[,*nmexp2*]

where *nmexp1* specifies a random access file buffer, *nmexp*=1,2,...,15

nmexp2 specifies the record number in the file, *nmexp2*=1,2,.., up to 335, depending on how much space is available on the disk; if *nmexp2* is omitted, the current record number is assumed.

This statement moves data from a file's buffer into a specified place in the file. Before PUTing data in a file, you must:

1) OPEN the file, thereby assigning a buffer and defining the access mode (must be R);
2) FIELD the buffer, so you can
3) place data into the buffer with LSET and RSET statements.

When BASIC encounters the statement:

PUT *nmexp,nmexp2*

it does the following:

- Gets the information needed to access the disk file
- Checks the access mode for this buffer (must be R)
- Acquires more disk space for the file if necessary to accommodate the record indicated by *nmexp2*
- Copies the buffer contents into the specified record of the disk file
- Updates the current record number to equal *nmexp2+1*

The "current record" is the record whose number is one higher than the last record accessed. The first time you access a file via a particular buffer, the current record is set equal to 1.

If the record number you PUT is higher than the end-of-file record number, then *nmexp2* becomes the new end-of-file record number.

This has an important implication. When you PUT a record whose number exceeds the EOF record number, space is allocated on the disk to accommodate the new highest record number plus all lower-numbered records. For example,

PUT *nmexp*,336

will always produce a DISK FULL message, since TRSDOS attempts to find space for all records from 1 to 336 – and 335 is the maximum number of records available on a diskette.

Examples (assume a file named SAMPLE/BAS exists and that you have previously written 10 records to it, so that LOF=10):

Program statement	Effect
1000 OPEN"R",1,"SAMPLE/BAS"	Open SAMPLE/BAS for random address under buffer 1
1010 FIELD 1,.....	Prepare buffer
1020 LSET	Place data in buffer
1030 PUT 1	Copy buffer contents into current record (=#1)
1035 LSET	Place data in buffer
1040 PUT 1,30	Acquire disk space for records 2 through 30 and copy buffer contents into record 30; set LOF=30
1045 LSET	Place data in buffer
1050 PUT 1,25	Copy buffer contents into record 25
1055 LSET	Place data in buffer
1060 PUT 1	Copy buffer contents into current record (=#26)

LSET and RSET
(place data in a random buffer field)

LSET *var$* = *exp$* and RSET *var$* = *exp$*

where *var$* is a field name

exp$* contains the data to be placed in the buffer field named by *var$

These two statements let you place character-string data into fields previously set up by a FIELD statement.

For example, suppose NM$ and AD$ have been defined as field names for a random file buffer. NM$ has a length of 18 characters, and AD$ has a length of 25 characters.

Now we want to place the following information into the buffer fields so it can be written to disk:

name: JIM CRICKET, JR.
address: 2000 EAST PECAN ST.

This is accomplished with the two statements:

```
LSET NM$="JIM CRICKET, JR. "
LSET AD$="2000 EAST PECAN ST. "
```

This puts the data in the buffer as follows:

JIM␣CRICKET,JR.␣␣␣	2000␣EAST␣PECAN␣ST.␣␣␣␣␣␣
NM$	**AD$**

Note that filler spaces were placed to the right of the data strings in both cases. If we had used RSET instead of LSET statements, the filler spaces would have been placed on the left. This is the **only** difference between LSET and RSET.

For example:

```
RSET NM$="JIM CRICKET, JR. "
RSET AD$="2000 EAST PECAN ST. "
```

places data in the fields as follows:

␣␣␣JIM␣CRICKET,JR.	␣␣␣␣␣␣2000␣EAST␣PECAN␣ST.
NM$	**AD$**

If a string item is too large to fit in the specified buffer field, it is always truncated on the right. That is, the extra characters on the right are ignored.

CVD, CVI and CVS (restore string to numeric form)

CVD(*exp$*)

where *exp$* defines an eight character string; *exp$* is typically the name of a buffer-field containing a numeric string. If LEN(*exp$*)<8, an ILLEGAL FUNCTION CALL error occurs; if LEN(*exp$*)>8, only the first eight characters are used.

CVI(*exp$*)

where *exp$* defines a two-character string; *exp$* is typically the name of a buffer-field containing a numeric string. If LEN(*exp$*)<2, an ILLEGAL FUNCTION CALL error occurs; if LEN(*exp$*)>2, only the first two characters are used.

CVS(*exp$*)

where *exp$* defines a four-character string; *exp$* is typically the name of a buffer-field containing a numeric string. If LEN(*exp$*)<4, an ILLEGAL FUNCTION CALL error occurs; if LEN(*exp$*)>4, only the first four characters are used.

These functions let you restore data to numeric form after it is read from disk. Typically the data has been read by a GET statement, and is stored in a random access file buffer.

The functions CVD, CVI, CVS are inverses of MKD$, MKI$, and MKS$, respectively.

For example, suppose the name GROSSPAY$ references an eight-byte field in a random-access file buffer, and after GETting a record, GROSSPAY$ contains a MKD$ representation of the number 13123.38.

Then the statement:

```
PRINT CVD(GROSSPAY$)-TAXES
```

prints the result of the difference, 13123.38–TAXES. Whereas the statement:

```
PRINT GROSSPAY$-TAXES
```

will produce a TYPE MISMATCH error, since string values cannot be used in arithmetic expressions.

Using the same example, the statement

```
A#=CVD(GROSSPAY$)
```

assigns the numeric value 13123.38 to the double-precision variable A#.

EOF (end-of-file detector)

EOF(*nmexp*)

where *nmexp* specifies a file buffer,
nmexp=1,2,...,15

This function checks to see whether all characters up to the end-of-file marker have been accessed, so you can avoid INPUT PAST END errors during sequential input.

Assuming *nmexp* specifies an open file, then EOF(*nmexp*) returns 0 (false) when the EOF record has not yet been read, and –1 (true) when it has been read.

Examples:

```
IF EOF(5) THEN PRINT"END OF FILE"FILENM$
IF EOF(NM%) THEN CLOSE NM%
```

The following sequence of lines reads numeric data from DATA/TXT into the array A( ). When the last data character in the file is read, the EOF test in line 30 "passes", so the program branches out of the disk access loop, preventing an INPUT PAST END error from occurring. Also note that the variable I contains the number of elements input into array A( ).

```
5 DIM A(100) 'ASSUMING THIS IS A SAFE VALUE
10 OPEN "I",1,"DATA/TXT"
20 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1,A(I%)
50 I%=I%+1
60 GOTO 30
70 REM PROGRAM CONTINUES HERE AFTER DISK INPUT
```

LOF (get end-of-file record number)

LOF(*nmexp*)

where *nmexp* specifies a random access buffer
nmexp=1,2,...,15

This function tells you the number of the last, i.e., highest numbered, record in a file. It is useful for both sequential and random access.

For example, during random access to a pre-existing file, you often need a way to know when you've read the last valid record. LOF provides a way.

Examples:

```
10 OPEN "R",1,"UNKNOWN/TXT"
20 FIELD 1,255 AS A$
30 FORI%=1 TO LOF(1)
40 GET 1,I%
50 PRINT A$
60 NEXT
```

In line 30, LOF(1) specifies the highest record number to be accessed.

Note: If you attempt to GET record numbers beyond the end-of-file record, BASIC simply fills the buffer with hexadecimal zeroes, and no error is generated.

When you want to add to the end of a file, LOF tells you where to start adding:

```
100 I%=LOF(1)+1 'HIGHEST EXISTING RECORD
110 PUT 1,I%    'ADD NEXT RECORD
```

MKD$, MKI$ and MKS$ (convert data, numeric-to-string)

MKD$(*nmexp*)

where *nmexp* is evaluated as a double-precision number

MKI$(*nmexp*)

where *nmexp* is evaluated as an integer, –32768<=*nmexp*<32768; if *nmexp* exceeds this range, an ILLEGAL FUNCTION CALL error occurs; any fractional component in *nmexp* is truncated

MKS$(*nmexp*)

where *nmexp* is evaluated as a single-precision number

These functions change a number to a "string". Actually the byte values which make up the number are not changed; only one byte, the internal data-type specifier, is changed, so that numeric data can be placed in a string variable. (See **LEVEL II Reference Manual,** VARPTR Function, for details of internal number representation.)

That is:

MKD$ returns an eight-byte string
MKI$ returns a two-byte string
MKS$ returns a four-byte string

Examples:

```
ASC(MKI$(I%)) equals the lsb of I%, i.e., (I% AND 255)
ASC(RIGHT$(MKI$(I),1))=the msb of I%, i.e., INT(I%/256)

LSET AVG$=MKS$(0.123)
```

AVG$ would typically reference a four-byte random buffer field. Now it contains a representation of the single-precision number 0.123.

```
LSET TALLY$=MKI$(I%)
```

Field name TALLY$ would now contain a two-byte representation of the integer I%.

```
A$=MKI$(8/I)
```

A$ becomes a two-byte representation of the integer portion of 8/I. Any fractional portion is ignored. Note that A$ in this case is a normal string variable, not a buffer-field name.

Suppose BASEBALL/BAT (a non-standard file extension) has been opened for random access using buffer 2, and the buffer has been FIELDed as follows:

field:	NM$	YRS$	AVG$	HR$	AB$	ERNING$
length:	16	2	4	2	4	4

NM$ is intended to hold a character string; AVG$, AB$ and ERNING$, converted single-precision values; YR$ and HR$, converted integers.

Suppose we want to write the following data record:

SLOW LEARNER played 38 years ; lifetime batting average .123; career homeruns, 11; at bats, 32768; . . . , earnings –13.75.

Then we'd use the make-string functions as follows:

```
1000 LSET NM$="SLOW LEARNER"
1010 LSET YRS$=MKI$(38)
1020 LSET AVG$=MKS$(.123)
1030 LSET HR$=MKI$(11)
1040 LSET AB$=MKS$(32768)
1050 LSET ERNING$=MKS$(-13.75)
```

After this sequence, you can write SLOW LEARNER's information to disk with the PUT statement. When you read it back from disk with GET, you will need to restore the numeric data from string to numeric form, using CVI and CVS functions.

Sequential Access Techniques

Sequential input/output is the simplest way to store data in disk files and retrieve it into BASIC variables.

To write to disk, you open a file for sequential output, PRINT# the data, and close the file. To read the data back, you simply open the file for sequential access and INPUT# the data directly into BASIC variables – in the same order as the data was written onto the disk.

Sequential Output – An Example

Suppose we want to store a table of English-to-metric conversion constants:

English unit	Metric equivalent
1 inch	2.54001 centimeters
1 mile	1.60935 kilometers
1 acre	4046.86 sq. meters
1 cubic inch	0.01638716 liter
1 U.S. gallon	3.785 liters
1 liquid quart	0.9463 liter
1 lb (avoir)	0.45359 kilogram

First we decide what the data image is going to be. Let's say we want it to look like this:

english unit–>*metric unit*, *factor* <EN>

For example, the stored data would start out:

IN–>CM,₿2.54001₿ <EN>

The following program will create such a data file.

Note: <EN> represents a carriage return, hex OD.

```
10 OPEN"O",1,"METRIC/TXT"
20 FORI%=1 TO 7
30 READ UNIT$,FACTR
40 PRINT#1,UNIT$;",";FACTR
50 NEXT
60 CLOSE
70 DATA IN->CM,2.54001,MI->KM,1.60935,ACRE->SQ.M,4046.86
80 DATA CU.IN->LTR,1.638716E-2,GAL->LTR,3.785
90 DATA LIQ.QT->LTR,0.9463,LB->KG,0.45359
```

Line 10 creates a disk file named METRIC/TXT, and assigns buffer 1 for sequential output to that file. The extension /TXT is used because sequential output always stores the data as ASCII-coded text.

Note: If METRIC/TXT already exists, line 10 will cause all its data to be lost. Here's why: Whenever a file is opened for sequential output, the EOF marker is set to the beginning of the file. In effect, TRSDOS "forgets" that anything has ever been written beyond this point.

Line 40 prints the current contents of UNIT$ and FACTR to the file buffer. The disk-write won't actually take place until the buffer is filled or you close the file, whichever happens first. Since the string items do not contain delimiters, it is not necessary to print explicit quotes around them. The explicit comma is sufficient.

Line 60 closes the file. The EOF marker points to the end of the last data item, i.e., 0.45359, so that later, during input, DISK BASIC will know when it has read all the data.

Sequential Input – An Example

The following program reads the data from METRIC/TXT into two "parallel" arrays, then asks you to enter a conversion problem.

```
5 CLEAR 500
10 DIM UNITS$(9),FACTR(9) 'ALLOWS FOR UP TO 10 DATA PAIRS
20 OPEN"I",1,"METRIC/TXT"
25 I%=0
30 IF EOF(1) THEN 70
40 INPUT#1,UNIT$(I%),FACTR(I%)
50 I%=I%+1
60 GOTO 30
70 REM..THE CONVERSION FACTORS HAVE BEEN READ IN
100 CLS: PRINT TAB(5)"*** ENGLISH TO METRIC CONVERSIONS ***"
110 FOR ITEM%=0TOI%-1
120 PRINT USING"(## )          %        %";ITEM%,UNIT$(ITEM%)
130 NEXT
140  PRINT@704,"WHICH CONVERSION ";
150 INPUT CHOICE%
155 PRINT@768,"ENTER ENGLISH QUANTITY";
160 INPUT V
170 PRINT"THE METRIC EQUIVALENT IS"V*FACTR(CHOICE%)
180 INPUT"PRESS ENTER TO CONTINUE";X
190 PRINT@704,CHR$(31); 'CLEAR TO END OF FRAME
200 GOTO 140
```

Line 20 opens the file for sequential input. The read pointer is automatically set to the beginning of the file.

Line 30 checks to see that the end-of-file record hasn't been read. If it has, control branches from the disk input loop to the part of the program that uses the newly acquired data.

Line 40 reads a value into the string array UNIT$( ), and a number into the single-precision array FACTR( ). Note that this INPUT list parallels the PRINT# list that created the data file (see the section "Sequential Output: An example"). This parallelism is not required, however. We could just as successfully have used:

```
40 INPUT#1,UNIT$(I%): INPUT#1,FACTR(I%)
```

How to update a file

Suppose you want to add more entries into the English-Metric conversion file. You can't simply re-open the file for sequential output and PRINT# the extra data – that would immediately set the end-of-file marker to the beginning of the file, effectively destroying the file's previous contents. Do this instead:

1) Open the file for sequential input
2) Input the entire file and store it (typically in one or more arrays)
3) Close the file
4) Add your new entries to the data array, or correct existing entries
5) Re-open the file for sequential output
6) Output the updated data array to the file
7) Close the file

If the file is too large to fit in memory, update it this way:

1) Open the file for sequential input
2) Open another new data file for sequential output
3) Input a block of data and update the data as necessary
4) Output the data to the new file
5) Repeat steps 3 and 4 until all data has been read, updated, and output to the new file; then go to step 6
6) Close both files
7) Kill the old data file
8) Rename the new file (TRSDOS RENAME command) to the name of the old file.

Sequential LINE INPUT - An Example

Using the line-oriented input, you can write programs that edit other BASIC program files : renumber them, change LPRINTs to PRINTs, etc. – as long as these "target" programs are stored in ASCII format.

The following program counts the number of lines in any disk file with the extension "/TXT".

```
10 CLEAR 300
20 INPUT "WHAT IS THE NAME OF THE PROGRAM"; PROG$
30 IF INSTR(PROG$,"/TXT")=0 THEN 110 'REQUIRE /TXT EXTENSION
40 OPEN"I",1,PROG$
50 I%=0
60 IF EOF(1)THEN 90
70 I%=I%+1: LINE INPUT#1,TEMP$
80 GOTO60
90 PRINT"THE PROGRAM IS"I%"LINES LONG."
100 CLOSE: GOTO20
110 PRINT "FILESPEC MUST INCLUDE THE EXTENSION '/TXT'"
120 GOTO20
```

For BASIC programs stored in ASCII, each program line ends with an < EN > character not preceded by an < LF > line feed. So the LINE INPUT in line 70 automatically reads one entire line at a time, into the variable TEMP$. Variable I% actually does the counting.

To try out the program, save DISKDUMP/BAS as a text file:

```
LOAD"DISKDUMP/BAS"  ENTER
SAVE"DISKDUMP/TXT",A ENTER
```

This gives you a second, ASCII-format version of DISKDUMP.

Now type in the line-counter program and tell it to examine the program DISKDUMP/TXT.

Disk Storage during Sequential Access

One thing that makes sequential access so simple is that you can generally ignore the details of disk storage. You just write your data and read it back.

Described below are a few of the technical details and hints you should keep in the back of your mind. In some situations, they will become important.

1. PRINT# statements don't write data directly to the disk; instead, the data is placed in the 256-byte output buffer. When this buffer is filled, the contents are automatically written to disk. (Closing the file will also write the buffer to disk.)

2. If a DISK FULL ERROR occurs during execution of a PRINT# statement, you should realize that the current contents of the output buffer have not been written to the file. The data in the disk file is intact, but it doesn't contain the last few values you PRINTed to it.

   If your variables still contain the data, you can recover it directly.

Random Access Techniques

Random access offers several advantages over sequential access:

- Instead of having to start reading at the beginning of a file, you can read any record you specify.
- To update a file, you don't have to read in the entire file, update the data, and write it out again. You can rewrite or add to any record you choose, without having to go through any of the other records.
- Random access is more efficient – data takes up less space and is read and written faster.
- Opening a file for random access allows you to write to and read from the file via the same buffer.
- Random access provides many powerful statements and functions to structure your data. Once you have set up the structure, random input/output becomes quite simple.

The last advantage listed above is also the "hard part" of random access. It takes a little extra thought.

For the purposes of random access, you can think of a disk file as a set of boxes – like a wall of post-office boxes. Just like the post office receptacles, the file boxes are numbered.

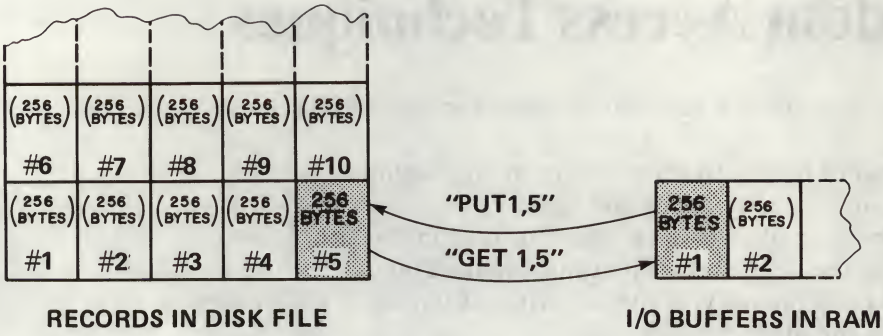
The number of boxes in a file will vary, but it's always a multiple of 5.

The smallest non-empty file contains 5 boxes, numbered 1 through 5. When the file needs more space to hold more data, TRSDOS provides it in increments of 5.

These fixed-sized boxes are referred to as "records". Each record contains 256 bytes, 255 of which are available for storing your data.

You can place data in any record, or read the contents of any record, with statements like:

```
PUT 1,5     write buffer-1 contents to record 5
GET 1,5     read the contents of record 5 into buffer-1
```



The buffer is a waiting area for the data. Before writing data to a file, you must place it in the buffer assigned to the file. After reading data from a file, you must retrieve it from the buffer.

As you can see from the sample PUT and GET statements above, data is passed to and from the disk in 256-byte chunks.

"That's a lot of data." But most values occupy only a few bytes:

Integers	2
Single-precision numbers	4
Double precision numbers	8
Strings	Up to 255

Therefore you'll want to place several values into the buffer before PUTting its contents into the disk file, to avoid wasting disk space.

This is accomplished by 1) dividing the buffer up into fields and naming them, then 2) placing the string or numeric data into the fields.

For example, suppose we want to store a glossary on disk. Each record will consist of a word followed by its definition. We start with:

```
100 OPEN"R",1,"GLOSSARY/BAS"
110 FIELD 1,15 AS WD$,240 AS MEANING$
```

Line 100 opens a file named GLOSSARY/BAS (creates it if it doesn't already exist); and gives buffer 1 random access to the file.

Line 110 defines two fields onto buffer 1:
WD$ consists of the first 15 bytes of the buffer;
MEANING$ consists of the last 240 bytes.

WD$ and MEANING$ are now **field-names.**

What makes field names different? Most string variables point to an area in memory called the string space. This is where the value of the string is stored.

Field names, on the other hand, point to the buffer area assigned in the FIELD statement. So, for example, the statement:

```
10 PRINT WD$ ": " MEANING$
```

displays the contents of the two buffer fields defined above.

These values are meaningless unless we first place data in the buffer. LSET, RSET and GET can all be used to accomplish this function. We'll start with LSET and RSET, which are used in preparation for disk output.

Our first entry is the word "left-justify" followed by its definition.

```
100 OPEN"R",1,"GLOSSARY/BAS"
110 FIELD 1,15 AS WD$,240 AS MEANING$
120 LSET WD$="LEFT-JUSTIFY"
130 LSET MEANING$="TO PLACE A VALUE IN A FIELD FROM LEFT
TO RIGHT; IF THE DATA DOESN'T FILL THE FIELD, BLANKS ARE ADDED
ON THE RIGHT; IF THE DATA IS TOO LONG, THE EXTRA CHARACTERS ON
THE RIGHT ARE IGNORED. LSET IS A LEFT-JUSTIFY FUNCTION."
```

Line 120 left-justifies the value in quotes into the first field in buffer 1. Line 130 does the same thing to its quoted string. When typing in line 130, you should insert line-feed <LF> characters (press the down arrow) to force line breaks as above. This makes it easier to print out the data after reading it back in to a string variable.

Note: RSET would place filler-blanks to the **left** of the item. Truncation would still be on the right.

Now that the data is in the buffer, we can write it to disk with a simple PUT statement:

```
140 PUT 1,1
150 CLOSE
```

This writes the first record into the file GLOSSARY/BAS.

To read and print the first record in GLOSSARY/BAS, use the following sequence:

```
160 OPEN"R",1,"GLOSSARY/BAS"
170 FIELD 1, 15 AS WD$,240 AS MEANING$
180 GET 1,1
190 PRINT WD$ ":  " MEANING$
200 CLOSE
```

Lines 160 and 170 are required only because we closed the file in line 150. If we hadn't closed it, we could go directly to line 180.

Random Access: A general procedure

The above example shows the necessary sequences to read and write using random access. But it does not demonstrate the primary advantages of this form of access – in particular, it doesn't show how to update existing files by going directly to the desired record.

The program below, GLOSSACC/BAS, develops the glossary example to show some of the techniques of random access for file maintenance. But before looking at the program, study this general procedure for creating and maintaining files via random access.

Step Number	See GLOSSACC/BAS, Line Number
1. OPEN the file	110
2. FIELD the buffer	120
3. GET the record to be updated	140
4. Display current contents of the record (use CVD,CVI,CVS before displaying numeric data)	145-170
5. LSET and RSET new values into the fields (use MKD$,MKI$,MKS$ with numeric data before setting it into the buffer)	210-230
6. PUT the updated record	240
7. To update another record, continue at step 3. Otherwise, go to step 8.	250-260
8. Close the file	270

```
10 REM. . GLOSSACC/BAS. . .
100 CLS: CLEAR 300
110 OPEN"R",1,"GLOSSARY/BAS"
120 FIELD 1,25 AS WD$,228 AS MEANING$,2 AS NX$
130 INPUT"WHAT RECORD DO YOU WANT TO ACCESS";R%
140 GET 1,R%
145 NX%=CVI(NX$)   'SAVE LINK TO NEXT ALPHABETICAL ENTRY
150 PRINT"WORD:  "WD$
160 PRINT"DEF'N:": PRINTMEANING$
170 PRINT"NEXT ALPHABETICAL ENTRY: RECORD#"NX%: PRINT
180 W$="": INPUT"TYPE NEW WORD<EN> OR <EN> IF OK";W$
190 D$="":PRINT"TYPE NEW DEF'N<EN> OR <EN> IF OK?":LINEINPUTD$
200 INPUT"TYPE NEW SEQUENCE NUMBER OR <EN> IF OK";NX%
210 IF W$<>""THEN LSET WD$=W$
220 IF D$<>""THEN LSET MEANING$=D$
230 LSET NX$=MKI$(NX%)
240 PUT 1,R%
245 R%=NX%  'USE NEXT ALPHA. LINK AS DEFAULT FOR NEXT RECORD
250 CLS: INPUT"    TYPE<EN> TO READ NEXT ALPHA. ENTRY,
    OR RECORD # <EN> FOR SPECIFIC ENTRY,
    OR 0 <EN> TO QUIT";R%
260 IF 0<R% THEN  140
270 CLOSE
280 END
```

Notice we've added a field, NX$, to the record (line 120). NX$ will contain the number of the record which comes next in alphabetical sequence. This enables us to proceed alphabetically through the glossary, provided we know which record contains the entry which should come first.

For example, suppose the glossary contains:

record#	word (WD$)	defn,	pointer to next alpha. entry (NX$)
1	LEFT-JUSTIFY	. . .	3
2	BYTE	. . .	4
3	RIGHT-JUSTIFY	. . .	0
4	HEXADECIMAL	. . .	1

When we read record 2 (BYTE), it tells us that record 4 (HEXADECIMAL) is next, which then tells us record 1 (LEFT-JUSTIFY) is next, etc. The last entry, record 3 (RIGHT-JUSTIFY), points us to zero, which we take to mean "THE END".

Since NX$ will contain an integer, we have to first convert that number to a two-byte string representation, using MKI$ (line 230 above).

```
RUN"GLOSSACC/BAS" ENTER
```

```
WHAT RECORD DO YOU WANT TO ACCESS? 4 ENTER
WORD:  HEXIDECIMAL
DEF'N:
CAPABLE OF EXISTING IN ANY OF 16 STATES, E.G., THE  HEXADECIMAL
DIGITS 0,1,2,...,9,A,B,C,D,E,F.  HEXADECIMAL NUMBERS ARE STRINGS
OF HEXADECIMAL DIGITS.

NEXT ALPHABETICAL ENTRY: RECORD# 1

TYPE NEW WORD<EN> OR <EN> IF OK? HEXADECIMAL ENTER
TYPE NEW DEF'N<EN> OR <EN> IF OK? ENTER

TYPE NEW SEQUENCE NUMBER OR <EN> IF OK? ENTER
```

```
    TYPE<EN> TO READ NEXT ALPHA. ENTRY,
    OR RECORD # <EN> FOR SPECIFIC ENTRY,
    OR 0 <EN> TO QUIT? ENTER
WORD:  LEFT-JUSTIFY
DEF'N:
TO PLACE DATA IN A FIELD FROM LEFT TO RIGHT, ADDING BLANKS AS
NECESSARY ON THE RIGHT TO FILL THE FIELD.  ANY EXTRA CHARACTERS
ON THE RIGHT ARE IGNORED.

NEXT ALPHABETICAL ENTRY: RECORD# 3

TYPE NEW WORD<EN> OR <EN> IF OK? ENTER
TYPE NEW DEF'N<EN> OR <EN> IF OK? ENTER

TYPE NEW SEQUENCE NUMBER OR <EN> IF OK? 2 ENTER
```

The following program displays the glossary in alphabetical sequence:

```
300 REM. . . GLOSSOUT/BAS. . .
310 CLS: CLEAR 300
320 OPEN"R", 1, "GLOSSARY/BAS"
330 FIELD 1, 15 AS WD$, 238 AS MEANING$, 2 AS NX$
340 INPUT"WHICH RECORD IS FIRST ALPHABETICALLY"; N%
350 GET 1, N%
360 PRINT: PRINTWD$
370 PRINTMEANING$
380 N%=CVI(NX$)
390 INPUT"PRESS ENTER TO CONTINUE"; X
400 IF N%<>0 THEN 350
410 CLOSE
420 END
```

Sub-Records

In the glossary example, each entry required the full 255 bytes available in the buffer. Often this is not the case. When each information-unit fills only a part of the buffer, it is a good idea to define several identical sub-records on the buffer. That way you don't waste disk space by PUTting records which contain only a few bytes of useful information.

For example, suppose we want to store a mailing list, and each entry will consist of:

field	field length
name	18
address	25
city	14
state	2
last purchase amt.	4

Total length of entry: 63

Note: The last-purchase-amount will be a single-precision number. Such values require 4 bytes, therefore the field length is 4.

If we didn't care about wasting space on the disk, we could use the following statement:

```
FIELD 1, 18 AS NM$, 25 AS AD$, 14 AS CTY$, 2 AS ST$, 4 AS LP$
```

PUTting such a buffer would create a record consisting of 63 bytes of information followed by 255-63=192 unused bytes.

A more efficient approach fields the buffer into identical sub-records. In this case, we can create 255/63 = 4 sub-records plus only 3 wasted bytes at the end.

Instead of using a very long FIELD statement to explicitly assign each field, we re-field the buffer once for each sub-record, using a dummy string, STARTHERE$, to start each sub-record at the appropriate position in the buffer.

```
FOR I%=0 TO 3
      FIELD 1,(I%*63) AS STARTHERE$,18 AS NM$(I%),
      25 AS AD$(I%),14 AS CTY$(I%),2 AS ST$(I%),4 AS LP$(I%)
NEXT
```

The first time through the loop, STARTHERE$ will have a length of zero. Therefore NM$(0) will start at the first byte; AD$(0), at the 19th byte, etc.; LP$(0) will end at the 63rd byte.

The second time through the loop, STARTHERE$ will have a length of 63. Therefore NM$(1) will start at the 64th byte; AD$(1), at the 92nd byte, etc.; LPS(1) will end at the 126th byte.

And so forth, until the buffer is completely defined.

To place values in the subrecords of the buffer: assume our mailing list entries are stored in four arrays, N$( ),A$( ),C$( ),S$( ),LP( ).

Then we can fill the buffer with four entries as follows:

```
FOR I%=0TO3
       LSET NM$(I%)=N$(I%)
       LSET AD$(I%)=A$(I%)
       LSET CT$(I%)=C$(I%)
       LSET ST$(I%)=S$(I%)
       LSET LP$(I%)=MKS$(LP(I%))
NEXT
```

How to Access Sub-Records

Since each record in such a file will contain four sub-records, we need a way to pull out the sub-record we want. This requires that each sub-record have a unique number which can be related to the record which contains it.

For this example, suppose we have a printout of the entire mailing list, starting from the first sub-record in record 1 and going through to the last sub-record in the last record. We then number them sequentially, **starting with 1**.

The following formulas use this number (we'll call it a key-number) to determine exactly where the sub-record is in the file:

If the sub-record's key-number is KEY%, then

PR% = INT((KEY%–1)/4)+1

where PR% is the physical record that contains the sub-record, and

SR%= KEY% – 4*(PR%–1)

where SR% is the sub-record number inside the physical record. For example, suppose we want to access the entry with key number = 37 (i.e., the 37th entry). Then the physical record which contains it is:

INT((37–1)/4)+1 == > record 10

And its position in record 10 is:

37 – 4*(10–1)+1 == > sub-record number 1

A full working program for creating and manipulating a mailing list follows:

```
100 CLEAR 1000
110 OPEN"R",1,"MAIL/BAS"
120 CLS: INPUT"TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT";N%
130 IF N%=0 THEN CLOSE: END
140 INPUT"TYPE KEY NUMBER<EN> OR 0<EN>"; KEY%
150 IF KEY%=0 THEN 120
160 PR%=INT((KEY%-1)/4)+1
170 SR%=KEY%-4*(PR%-1)
180 FIELD 1,((SR%-1)*63) AS STARTHERE$,18 AS NM$, 25 AS AD$,
    14 AS CTY$,2 AS ST$,4 AS LP$
190 GET 1,PR%
200 IF N%=2THEN300
210 PRINT"WRITING SUBRECORD #"SR%"IN PHYSICAL RECORD #"PR%
220 PRINT: PRINT"NAME?"TAB(20);: LINEINPUT N$: LSET NM$=N$
230 PRINT"ADDRESS?"TAB(20);: LINEINPUT A$: LSET AD$=A$
240 PRINT"CITY?"TAB(20);: LINEINPUT C$: LSET CTY$=C$
250 PRINT"STATE?"TAB(20);: LINEINPUT S$: LSET ST$=S$
260 PRINT"LAST PURCHASE"TAB(20);:INPUTLP:LSEKS$(LP)
270 PUT 1,PR%: PRINT: INPUT"PRESS <EN> TO GO ON";X: GOTO 120
300 PRINT"READING SUBRECORD #"SR%"IN PHYSICAL RECORD #"PR%
310 PRINT: PRINT"NAME"TAB(20)NM$
320 PRINT"ADDRESS"TAB(20)AD$
330 PRINT"CITY"TAB(20)CTY$
340 PRINT"STATE"TAB(20)ST$
350 PRINT USING"LAST PURCHASE       $#####.##";CVS(LP$)
360 PRINT: INPUT"PRESS <EN> TO GO ON";X: GOTO120
```

This program actually doesn't require you to fill the buffer with four meaningful sub-records. As soon as you've placed a sub-record in the correct position in the field, the entire buffer is written to disk. However, the extra space is not wasted; it is always available for subsequent sub-records to be added.

Note that this would not be the most efficient way to create a list at one "sitting". In such a case you'd probably want to fill the buffer with four sub-records before doing the disk-write. The above program does, however, show you how to update a file using random access.

```
RUN"MAILACC/BAS" ENTER
```

```
TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT? 1 ENTER
TYPE KEY NUMBER<EN> OR 0<EN>? 3 ENTER
WRITING SUBRECORD # 3 IN PHYSICAL RECORD # 1

NAME?                DUNMAN, I.C. ENTER
ADDRESS?             2222 SECOND STREET ENTER
CITY?                OLD PORT ENTER
STATE?               AZ ENTER
LAST PURCHASE        ? 12.81 ENTER

PRESS <EN> TO GO ON? ENTER
```

```
TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT? 2 ENTER
TYPE KEY NUMBER<EN> OR 0<EN>? 1 ENTER
READING SUBRECORD # 1 IN PHYSICAL RECORD # 1

NAME                 JOHNSON, J.R.
ADDRESS              1024 RAM DRIVE
CITY                 FORT WUMPUS
STATE                TX
LAST PURCHASE        $ 188.75

PRESS <EN> TO GO ON? ENTER
```

```
TYPE 1<EN> TO WRITE, 2<EN> TO READ,
    0<EN> TO QUIT? 0 ENTER
READY
>_
```

Overlapping Fields

Suppose you want to access a field in two ways – in total and in part. Then you can assign two field names to the same area of the buffer.

For example, if the first two digits of a six-digit stock-number specify a category, you might use the following field structure:

```
FIELD 1, 6 AS STOCK$, .....
FIELD 1, 2 AS CTG$, .....
```

Now STOCK$ will reference the entire stock-number field, while CTG$ will reference only the first two digits of the number.

DISK BASIC Error Messages

Code	Message	Explanation
50	FIELD OVERFLOW	More than 255 bytes were allocated to a random-access buffer.
51	INTERNAL ERROR	Error in disk operating system itself, or disk I/O fault.
52	BAD FILE NUMBER	A file-buffer number was used improperly; number has not been assigned to a file with an OPEN statement.
53	FILE NOT FOUND	Attempt to read from a file which is not contained on the disk; check name/extension to see they were specified correctly.
54	BAD FILE MODE	Attempt to perform disk file input or output which conflicts with the mode in which the file was opened.
57	DISK I/O ERROR	An error occurred during data transfer between the Computer and a disk file.
61	DISK FULL	All available space on the diskette has been used.
62	INPUT PAST END	During sequential input to a variable, the end of file was reached before any data characters were read.
63	BAD RECORD NUMBER	Record number in a PUT statement exceeded the range <1,340>.
64	BAD FILENAME	An invalid file specification was provided; study "File Specification", **TRSDOS Overview**.

Note: Disk errors cannot be simulated via the ERROR statement

Code	Message	Explanation
66	DIRECT STATEMENT IN FILE	Attempt to LOAD, RUN, or MERGE a disk file which is not a BASIC program.
67	TOO MANY FILES	Attempt to place more than 48 files on a single diskette.
68	DISK WRITE-PROTECTED	Attempt to write to disk with write-protect notch covered.
69	FILE ACCESS DENIED	Attempt to access existing file with incorrect password.

Appendices



APPENDICES

Contents of This Section

Glossary 2
Memory Map 3
TRSDOS Character Tables 14
Base Conversions 18

Glossary

access

The method in which information is read from or written to disk; see random access and sequential access.

address

A location in memory, usually specified as a two-byte hexadecimal number. The address range<0 to FFFF>is represented in decimal as<0 to 32767> <-32768,. . . , -1>

alphabetic

Referring strictly to the letters A-Z.

alphanumeric

Referring to the set of letters A-Z and the numerals 0-9.

argument

The string or numeric quantity which is supplied to a function and is then operated on to derive a result; this result is referred to as the **value** of the function.

array

An organized set of elements which can be referenced in total or individually, using the array name and one or more subscripts. In BASIC, any variable name can be used to name an array; and arrays can have one or more dimensions. AR( ) signifies a one-dimensional array named AR; AR(,) signifies a two-dimensional array named AR; etc.

ASCII

American Standard Code for Information Interchange. This method of coding is used to store textual data. Numeric data is typically stored in a more compressed format.

ASCII format disk file

Disk files in which each byte corresponds to one character of the original data. For example, a BASIC program stored in ASCII format "looks like" the program listing, except that each character is ASCII-coded. Compare to compressed-format file.

background task

A relatively slow routine which the computer executes along with other background tasks, and which is subject to interrupts. When the interrupt-driven tasks are completed, the background task continues. See **foreground task, task.**

backup disk

An exact copy of the original: a "safe copy". You should keep backups of your original TRSDOS diskette and all important data diskettes.

BASIC

Beginners' All-purpose Symbolic Instruction Code, the programming language which is stored in ROM in the TRS-80. Radio Shack supports LEVEL I BASIC, LEVEL II BASIC, and DISK BASIC. LEVEL II is a subset of DISK BASIC.

baud

Signalling speed in bits per second. The LEVEL II cassette interface operates at 500 baud.

binary

Having two possible states, e.g., the binary digits 0 and 1. The binary (base 2) numbering system uses sequences of zeroes and ones to represent quantities. This is analagous to the Computer's internal representation of date, using electrical values for 0 and 1.

bit

Binary digit; the smallest unit of memory in the Computer, capable of representing the values 0 and 1.

bootstrap program

A fundamental or primitive program which takes the Computer from an OFF condition to one in which it is capable of loading and executing a higher-level program – i.e., a program which allows the Computer to pull itself up "by its own bootstraps". A program which initializes the Computer.

break

To interrupt execution of a program. In BASIC the statement

```
STOP
```

causes a break in execution, as does pressing the BREAK key.

buffer

An area in RAM where data is accumulated for further processing. For example, to pass data from BASIC to a disk file, and vice-versa, the data must go through a file-buffer.

buffer field

A portion of the buffer which you define as the storage area for a buffer-field variable. Dividing a buffer into fields allows you to pass multiple values to and from disk storage.

byte

The smallest addressable unit of memory in the Computer, consisting of 8 consecutive bits, and capable of representing 256 different values, e.g., decimal values from 0 to 255.

compressed-format

A method of storing information in less space than a standard ASCII representation would require. An integer always requires two bytes; a single-precision number, four; a double-precision number, 8 — regardless of how many characters are required to represent the numbers as text. String values cannot be stored in compressed format.

BASIC programs in RAM and non-ASCII disk files are stored in compressed-format, with all BASIC keywords stored as special one-byte codes.

command file

A TRSDOS disk file with the extension /CMD. Such a file should consist of an executable Z-80 program, since TRSDOS will load and attempt to execute it when you type:

filename **ENTER**

Command files can be placed on any disk; in effect, they extend the set of TRSDOS library commands (though, of course, they remain external to the TRSDOS system files).

close

Terminate access to a disk file. Before re-accessing the file, you must re-open it.

data

Information that is passed to our output from a program; under LEVEL II and DISK BASIC, there are four types of data:

- integer numbers
- single-precision floating point numbers
- double-precision floating point numbers
- character-string sequences, or just "strings"

data/device control block (DCB)

An area in RAM associated with an I/O buffer, containing information the Operating System requires in order to access the I/O device or file.

debug

To isolate and remove logical or syntax errors from a program.

decimal

Capable of assuming one of ten states, e.g., the decimal digits 0,1,...,9. Decimal (base 10) numbering is the everyday system, using sequences of decimal digits. Decimal numbers are stored in binary code in the Computer.

default

An action or value which is supplied by the Computer when you do not specify an action or value to be used.

delimiter

A character which marks the beginning or end of a data item, and is not a part of the data. For example, the double-quote symbol is a string delimiter to BASIC.

destination

The device or address which receives the data during a data transfer operation. For example, during a BACKUP operation, the destination disk is the one onto which the source-disk is being copied.

device

A physical part of the computer system used for data I/O, e.g., keyboard, display, line printer, cassette, disk drive, voice synthesizer.

directory

A listing of the files which are contained on a disk.

disk drive or **Mini Disk drive**

The physical device which writes data onto diskettes and retrieves it.

diskette or **disk**

A magnetic recording medium for mass data storage.

drive specification or **drivespec**

An optional field in a TRSDOS file specification and in some TRSDOS commands, consisting of a colon followed by one of the digits 0 through 3. The drivespec is used to specify which drive is to be used for a disk read or write.

When the drivespec is omitted from a command involving a read operation, TRSDOS will search through all the disks for the desired file, starting with drive 0.

When the drivespec is omitted from a command involving a write operation, TRSDOS will generally search through all non write-protected drives for the desired file.

drive number

An integer value from 0 to 3, specifying one of the Mini Disk drives. Drive 0 is closest to the Expansion Interface, and Drive 3 is farthest away. Drive 0 must always contain the TRSDOS diskette, with a couple of exceptions.

dummy variable

A variable name which is used in an expression to meet syntactic requirements, but whose value is insignificant to the programmer.

edit

To change existing information.

end of file or **EOF**

A marker which indicates the end of a disk file, i.e., where the meaningful data ends and the unknown begins.

entry point

The address of a machine-language program or routine where execution is to begin. This is not necessarily the same as the starting address. Entry point is also referred to as the **transfer address.**

expression

A meaningful sequence of one or more variables, constants, operators and functions.

field

A user-defined subdivision of a random access file-buffer, created and named with the FIELD statement.

field name

A string variable which has been assigned to a field in a random access file-buffer via the FIELD statement.

file

An organized collection of related data. Under TRSDOS, a file is the largest block of information which can be addressed with a single command. BASIC programs and data sets are stored on disk in distinct files.

file extension

An optional field in a file specification, consisting of a / followed by one alphabetic and up to two alphanumeric characters; the extension can be used to identify the file type, e.g., /BAS, /TXT, /CIM, for BASIC, text, and core image, respectively.

filename

A required field in a file specification, consisting of one alphabetic followed by up to 7 alphanumeric characters. Filenames are assigned when a file is created or renamed.

file specification or **filespec**

A sequence of characters which specifies a particular disk file under TRSDOS, consisting of a mandatory filename, followed by an optional extension, password, and drivespec.

foreground task

A relatively fast routine which the Computer must execute periodically, in sequence with other foreground tasks. Such tasks are interrupt-driven. See **background task, task, interrupt.**

format

To organize a new or magnetically erased diskette into tracks and sectors, via the TRSDOS FORMAT utility. BACKUP also implicitly formats a blank diskette. Formatted diskettes contain 35 tracks, each of which contains 10 sectors.

granule

The smallest unit of allocatable space on a disk, consisting of 5 sectors.

hexadecimal or **hex**

Capable of existing in one of 16 possible states. For example, the hexadecimal digits are 0,1,2,. . ,,9,A,B,C,D,E,F. Hexadecimal (base-16) numbers are sequences of hexadecimal digits. Address and byte values are frequently given in hexadecimal form. Under DISK BASIC, hexadecimal constants can be entered by prefixing the constant with &H.

increment

The value which is added to a counter each time one cycle of a repetitive procedure is completed.

input

To transfer data from outside the Computer (from a disk file, keyboard, etc.) into RAM.

interrupt

A signal which causes the Computer to interrupt whatever it is doing and perform some other specified task; when the task is completed, the Computer will generally resume execution of the previous task. The TRS-80 Expansion Interface includes a 25 millisecond "heartbeat" interrupt, which is used to drive the real-time clock and other foreground tasks. Interrupt-driven tasks can be scheduled and assigned priorities, so that the Computer appears to be doing two or more things "at once".

kilobyte or **K**

1024 bytes of memory. Thus a 12 K ROM includes 12*1024=12288 bytes.

library commands

A set of overlayed TRSDOS commands which are overlayed as needed into RAM between 5200 and 6FFF, to see which library commands are available, use the TRSDOS LIB command:

LIB <EN>

logical expression

An expression which is evaluated as either True (=-1) or FALSE (=0).

logical record

A block of data which contains from 1 to 256 bytes, and can be addressed as a unit, regardless of whether the logical record is contained in a single record or spans two physical records.

machine language

The Z-80 instruction set, usually specified in hexadecimal code. All higher-level languages must be translated into machine-language in order to be executed by the Computer.

null string

A string which has a length of zero; For example, the assignment

A$ = " "

makes A$ a null-string.

object code

Machine language derived from "source code", typically, from Assembly Language.

octal

Capable of existing in one of 8 states, for example, the octal digits are 0,1,...,7. Octal (base-8) numbers are sequences of octal digits. Address and byte values are frequently given in octal form. Under DISK BASIC, an octal constant can be entered by prefixing the octal number with the symbol &O.

open

To prepare a file for access by assigning a sequential input, sequential output, or random I/O buffer to it.

output

To transfer data from inside a Computer's memory to some external area, e.g., a disk file or a line printer.

overlay

To replace one block of code in RAM with another block. Also, the code which replaces the previous contents of RAM. For example, the TRSDOS system routines are stored on disk and loaded into a common area of RAM as **overlays.**

parameter

Optional information supplied with a command to specify how the command is to operate. TRSDOS parameters are placed inside parentheses.

password

An optional field in a filespec consisting of one alphanumeric followed by up to 7 additional alphanumeric characters. If a file is created without a password, 8 blanks become the default password. To access a file, you must specify the password in the filespec.

Using the TRSDOS ATTRIB command, you can assign both update and access passwords; the access password will grant only a limited degree of access, while the update password grants total access to the file. See **filespec.**

physical record

The smallest amount of data which can be written to a disk file or read from it; under TRSDOS, physical records consist of 256 bytes. Note that physical record length can be ignored by the assembly-language programmer, since TRSDOS supports logical records of from 1 to 256 bytes in length.

prompt

A character or message provided by the Computer to indicate that it's ready to accept keyboard input.

protected file

A disk file which has a non-blank password, and therefore can only be accessed by reference to that password.

protection level

The degree of access granted by using the access password: kill, rename, write, read, or execute.

random access memory or **RAM**

Semiconductor memory which can be addressed directly and either read from or written to. "User RAM" is that portion of RAM which is left untouched by TRSDOS and DISK BASIC code, from hex 7000 to end of memory.

real-time clock

An interrupt driven routine that keeps time by updating certain memory locations every 25 milliseconds, regardless of what the current background task is. At power-on, the real-time clock is set to 00:00:00. When interrupts are disabled, the clock is stopped.

reset

To press the reset button on the rear left of the TRS-80, next to the Expansion Interface connection. Pressing reset is equivalent to powering up the Computer, except that the contents of user RAM are unaffected.

resident system program

That part of TRSDOS which remains in RAM; the "executive TRSDOS program", which calls in other TRSDOS code as needed.

read-only memory or **ROM**

Pre-programmed semiconductor memory which is directly addressable but can only be read, not written to. The LEVEL II TRS-80 includes 12K of ROM, where a bootstrap program, LEVEL II BASIC, and other code are permanently stored.

routine

A sequence of instructions to carry out a certain function; typically, a routine may be called from multiple points in a program. For example: keyboard scan routine.

sector

One-tenth of a track on a diskette, containing 256 bytes of storage; a TRSDOS "physical record".

sequential access

Reading from a disk file or writing to it "from start to finish", without being able to directly access a particular record in the file.

statement

A complete instruction in BASIC.

string

Any sequence of characters which must be examined verbatim for meaning: in other words, the string does not correspond to a quantity. For example, the *number* 1234 represents the same quantity as 1000+234, but the *string* "1234" does not. (String addition is actually concatenation, or stringing-together, so that: "1234" equals "1" + "2" + "3" + "4").

system file

A TRSDOS disk file with the extension /SYS. Such files are read-protected. To avoid confusion, don't use the extension /SYS on your own disk files.

syntax

The "grammatical" requirements for a command or statement. Syntax generally refers to punctuation and ordering of elements within a statement. See "Notation Conventions", **General Information**, for a description of syntax abbreviations used in this manual.

task

A relatively fundamental routine which the Computer performs periodically or upon request.

track

One of 35 concentric circles on the disk, each of which contains 10 sectors, or 2560 bytes of storage. The tracks are not physical entities like grooves on a record; they are magnetic traces.

transfer address

See **entry point**.

TRSDOS

TRS-80 Disk Operating System, pronounced "triss-doss". TRSDOS is supplied on disk and is then loaded into RAM.

user RAM or **user memory**

See **random access memory.**

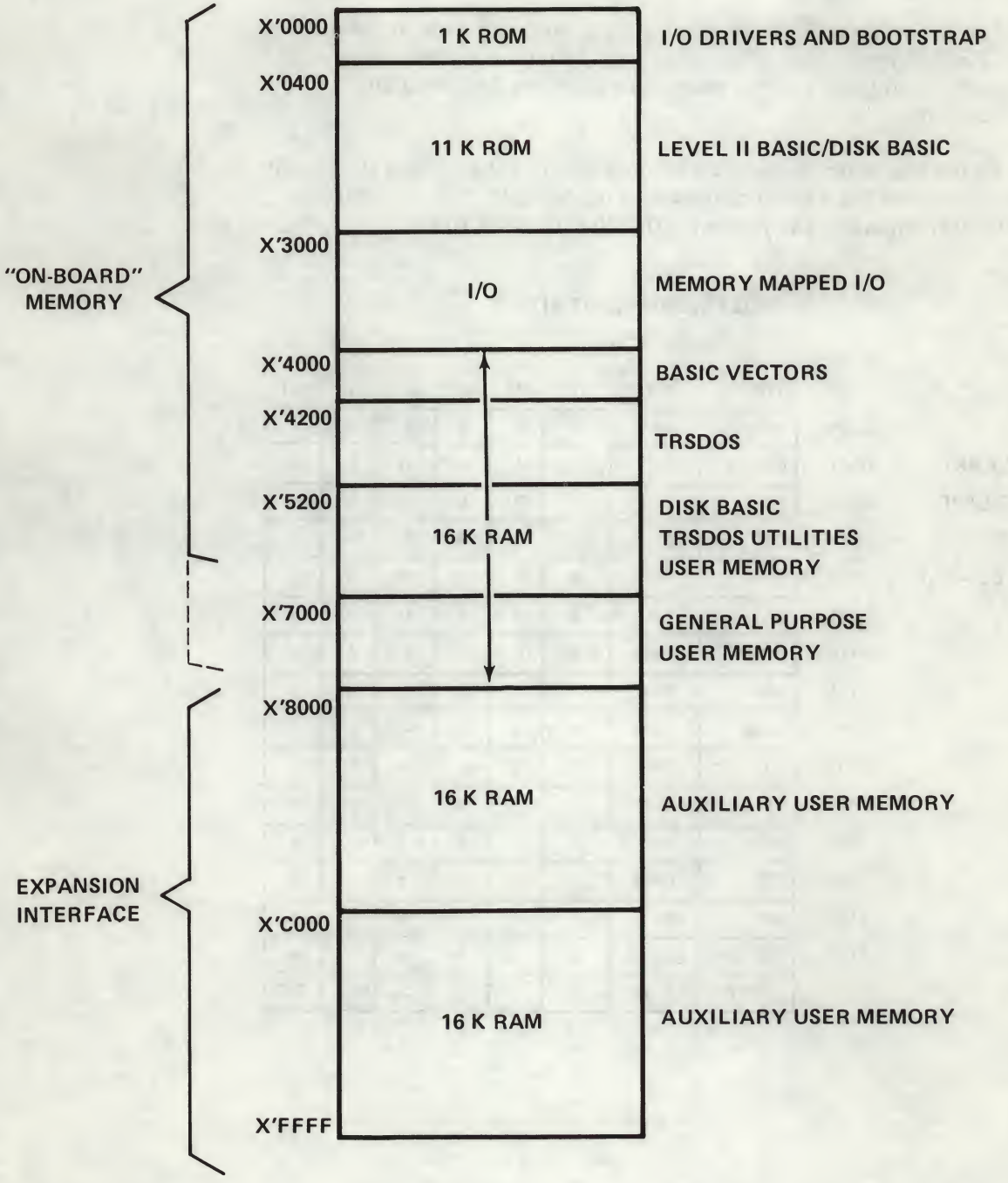
utility

A program or routine which serves a limited, specific purpose. There are two extended TRSDOS utilities, FORMAT and BACKUP, and two non-TRSDOS utilities, DISKDUMP/BAS and TAPEDISK.

write-protect

To physically protect a disk from being written to by placing a tape over the write-protect notch.

Memory Map


X'0000
1 K ROM
I/O DRIVERS AND BOOTSTRAP
X'0400
11 K ROM
LEVEL II BASIC/DISK BASIC
X'3000
I/O
MEMORY MAPPED I/O
"ON-BOARD" MEMORY
X'4000
BASIC VECTORS
X'4200
TRSDOS
X'5200
16 K RAM
DISK BASIC
TRSDOS UTILITIES
USER MEMORY
X'7000
GENERAL PURPOSE
USER MEMORY
X'8000
16 K RAM
AUXILIARY USER MEMORY
EXPANSION INTERFACE
X'C000
16 K RAM
AUXILIARY USER MEMORY
X'FFFF

TRSDOS Character Tables

Bit-Pattern Codes

The following table illustrates the bit pattern for each of the 128 TRSDOS characters. The remaining 128 codes represent special graphics and space compression characters, as described later. See Notes.

To use the table: Combine the most significant and least significant bit-patterns for a given character. For example, the character Q is represented by the pattern: 1010001 (decimal 81).

MOST SIGNIFICANT BITS
($b_7 - b_5$)

LEAST SIGNIF. BITS ($b_4 - b_1$)	000	001	010	011	100	101	110	111
0000	NULL	DLE	SP	0	@	P	@	p
0001	BREAK	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BKSP	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	↑	k	↑
1100	FF	HOME	,	<	L	↓	l	↓
1101	CR	BOL	—	=	M	←	m	←
1110	CURON	EREOL	.	>	N	→	n	→
1111	CUROFF	EREOF	/	?	O	_	o	DEL

Decimal/Hexadecimal Codes

Code			Code			Code		
Dec.	Hex.	Char.	Dec.	Hex.	Char.	Dec.	Hex.	Char.
0	00	NULL	32	20	SPACE	64	40	@
1	01	BREAK	33	21	!	65	41	A
2	02	STX	34	22	"	66	42	B
3	03	ETX	35	23	#	67	43	C
4	04	EOT	36	24	$	68	44	D
5	05	ENQ	37	25	%	69	45	E
6	06	ACK	38	26	&	70	46	F
7	07	BEL	39	27	'	71	47	G
8	08	BKSP	40	28	(	72	48	H
9	09	HT	41	29	)	73	49	I
10	0A	LF	42	2A	*	74	4A	J
11	0B	VT	43	2B	+	75	4B	K
12	0C	FF	44	2C	,	76	4C	L
13	0D	CR	45	2D	—	77	4D	M
14	0E	CURON	46	2E	.	78	4E	N
15	0F	CUROFF	47	2F	/	79	4F	O
16	10	DLE	48	30	0	80	50	P
17	11	DC1	49	31	1	81	51	Q
18	12	DC2	50	32	2	82	52	R
19	13	DC3	51	33	3	83	53	S
20	14	DC4	52	34	4	84	54	T
21	15	NAK	53	35	5	85	55	U
22	16	SYN	54	36	6	86	56	V
23	17	ETB	55	37	7	87	57	W
24	18	CAN	56	38	8	88	58	X
25	19	EM	57	39	9	89	59	Y
26	1A	SUB	58	3A	:	90	5A	Z
27	1B	ESC	59	3B	;	91	5B	↑
28	1C	HOME	60	3C	<	92	5C	↓
29	1D	BOL	61	3D	=	93	5D	←
30	1E	EREOL	62	3E	>	94	5E	→
31	1F	EREOF	63	3F	?	95	5F	—

Note: 96-127 (hex 60-7F) are lower-case counterparts to 64-95 (hex 40-5F).; only upper-case characters are displayable.

Notes

The TRSDOS character set may be subdivided into the following functional groups:

decimal code	hex code	function
0-31	00-1F	Control characters
32-95	20-5F	Keyboard/display characters
96-127	60-7F	Non-printing characters (*code-32* is printed)
128-191	80-BF	Graphics characters
192-255	C0-FF	Space-compression codes

The following **control characters** may be entered directly from the keyboard:

character	
BREAK	BREAK
BKSP	←
HT	→
LF	↓
CR	ENTER
CAN	SHIFT ←
EM	SHIFT →
SUB	SHIFT ↓
ESC	SHIFT ↑
EREOF	CLEAR
SP	SPACE-BAR

For a description of the **graphics characters**, run the following program. If you do not have a line printer connected, change all LPRINTs to PRINTs and use the shift-@ key to pause the display.

```
10 CLS: DEFINT A-Z
20 FORI=128 TO 191
30 POKE 15360,I
35 LPRINT CHR$(138)
40 LPRINT"GRAPHICS CODE # ";I
45 LPRINT CHR$(138)
50 A1=POINT(0,0): A2=POINT(1,0)
60 A3=POINT(0,1): A4=POINT(1,1)
70 A5=POINT(0,2): A6=POINT(1,2)
80 LPRINTTAB(8)CHR$(A1*(-40)+48);CHR$(A2*(-40)+48)
90 LPRINTTAB(8)CHR$(A3*(-40)+48);CHR$(A4*(-40)+48)
100 LPRINTTAB(8)CHR$(A5*(-40)+48);CHR$(A6*(-40)+48)
110 NEXT
```

The **space-compression codes** provide a compact means of representing strings of blanks from zero to 63 blanks. For example, C0 represents zero blanks; C1, 1 blank; C2, 2 blanks; FF, 63 blanks.

Base Conversions

The following table lists base conversions for all one-byte values.

DEC.	BINARY	HEX.	OCT.
0	00000000	00	000
1	00000001	01	001
2	00000010	02	002
3	00000011	03	003
4	00000100	04	004
5	00000101	05	005
6	00000110	06	006
7	00000111	07	007
8	00001000	08	010
9	00001001	09	011
10	00001010	0A	012
11	00001011	0B	013
12	00001100	0C	014
13	00001101	0D	015
14	00001110	0E	016
15	00001111	0F	017
16	00010000	10	020
17	00010001	11	021
18	00010010	12	022
19	00010011	13	023
20	00010100	14	024
21	00010101	15	025
22	00010110	16	026
23	00010111	17	027
24	00011000	18	030
25	00011001	19	031
26	00011010	1A	032
27	00011011	1B	033
28	00011100	1C	034
29	00011101	1D	035
30	00011110	1E	036
31	00011111	1F	037
32	00100000	20	040
33	00100001	21	041
34	00100010	22	042
35	00100011	23	043
36	00100100	24	044
37	00100101	25	045
38	00100110	26	046
39	00100111	27	047
40	00101000	28	050
41	00101001	29	051
42	00101010	2A	052
43	00101011	2B	053
44	00101100	2C	054
45	00101101	2D	055
46	00101110	2E	056
47	00101111	2F	057
48	00110000	30	060
49	00110001	31	061
50	00110010	32	062
51	00110011	33	063
52	00110100	34	064
53	00110101	35	065
54	00110110	36	066
55	00110111	37	067
56	00111000	38	070
57	00111001	39	071
58	00111010	3A	072
59	00111011	3B	073
60	00111100	3C	074
61	00111101	3D	075
62	00111110	3E	076
63	00111111	3F	077
64	01000000	40	100
65	01000001	41	101
66	01000010	42	102
67	01000011	43	103
68	01000100	44	104
69	01000101	45	105
70	01000110	46	106
71	01000111	47	107
72	01001000	48	110
73	01001001	49	111
74	01001010	4A	112
75	01001011	4B	113
76	01001100	4C	114
77	01001101	4D	115
78	01001110	4E	116
79	01001111	4F	117
80	01010000	50	120
81	01010001	51	121
82	01010010	52	122
83	01010011	53	123
84	01010100	54	124
85	01010101	55	125

DEC.	BINARY	HEX.	OCT.
86	01010110	56	126
87	01010111	57	127
88	01011000	58	130
89	01011001	59	131
90	01011010	5A	132
91	01011011	5B	133
92	01011100	5C	134
93	01011101	5D	135
94	01011110	5E	136
95	01011111	5F	137
96	01100000	60	140
97	01100001	61	141
98	01100010	62	142
99	01100011	63	143
100	01100100	64	144
101	01100101	65	145
102	01100110	66	146
103	01100111	67	147
104	01101000	68	150
105	01101001	69	151
106	01101010	6A	152
107	01101011	6B	153
108	01101100	6C	154
109	01101101	6D	155
110	01101110	6E	156
111	01101111	6F	157
112	01110000	70	160
113	01110001	71	161
114	01110010	72	162
115	01110011	73	163
116	01110100	74	164
117	01110101	75	165
118	01110110	76	166
119	01110111	77	167
120	01111000	78	170
121	01111001	79	171
122	01111010	7A	172
123	01111011	7B	173
124	01111100	7C	174
125	01111101	7D	175
126	01111110	7E	176
127	01111111	7F	177
128	10000000	80	200
129	10000001	81	201
130	10000010	82	202
131	10000011	83	203
132	10000100	84	204
133	10000101	85	205

DEC.	BINARY	HEX.	OCT.
134	10000110	86	206
135	10000111	87	207
136	10001000	88	210
137	10001001	89	211
138	10001010	8A	212
139	10001011	8B	213
140	10001100	8C	214
141	10001101	8D	215
142	10001110	8E	216
143	10001111	8F	217
144	10010000	90	220
145	10010001	91	221
146	10010010	92	222
147	10010011	93	223
148	10010100	94	224
149	10010101	95	225
150	10010110	96	226
151	10010111	97	227
152	10011000	98	230
153	10011001	99	231
154	10011010	9A	232
155	10011011	9B	233
156	10011100	9C	234
157	10011101	9D	235
158	10011110	9E	236
159	10011111	9F	237
160	10100000	A0	240
161	10100001	A1	241
162	10100010	A2	242
163	10100011	A3	243
164	10100100	A4	244
165	10100101	A5	245
166	10100110	A6	246
167	10100111	A7	247
168	10101000	A8	250
169	10101001	A9	251
170	10101010	AA	252
171	10101011	AB	253
172	10101100	AC	254
173	10101101	AD	255
174	10101110	AE	256
175	10101111	AF	257
176	10110000	B0	260
177	10110001	B1	261
178	10110010	B2	262
179	10110011	B3	263
180	10110100	B4	264
181	10110101	B5	265
182	10110110	B6	266

DEC.	BINARY	HEX.	OCT.
183	10110111	B7	267
184	10111000	B8	270
185	10111001	B9	271
186	10111010	BA	272
187	10111011	BB	273
188	10111100	BC	274
189	10111101	BD	275
190	10111110	BE	276
191	10111111	BF	277
192	11000000	C0	300
193	11000001	C1	301
194	11000010	C2	302
195	11000011	C3	303
196	11000100	C4	304
197	11000101	C5	305
198	11000110	C6	306
199	11000111	C7	307
200	11001000	C8	310
201	11001001	C9	311
202	11001010	CA	312
203	11001011	CB	313
204	11001100	CC	314
205	11001101	CD	315
206	11001110	CE	316
207	11001111	CF	317
208	11010000	D0	320
209	11010001	D1	321
210	11010010	D2	322
211	11010011	D3	323
212	11010100	D4	324
213	11010101	D5	325
214	11010110	D6	326
215	11010111	D7	327
216	11011000	D8	330
217	11011001	D9	331
218	11011010	DA	332

DEC.	BINARY	HEX.	OCT.
219	11011011	DB	333
220	11011100	DC	334
221	11011101	DD	335
222	11011110	DE	336
223	11011111	DF	337
224	11100000	E0	340
225	11100001	E1	341
226	11100010	E2	342
227	11100011	E3	343
228	11100100	E4	344
229	11100101	E5	345
230	11100110	E6	346
231	11100111	E7	347
232	11101000	E8	350
233	11101001	E9	351
234	11101010	EA	352
235	11101011	EB	353
236	11101100	EC	354
237	11101101	ED	355
238	11101110	EE	356
239	11101111	EF	357
240	11110000	F0	360
241	11110001	F1	361
242	11110010	F2	362
243	11110011	F3	363
244	11110100	F4	364
245	11110101	F5	365
246	11110110	F6	366
247	11110111	F7	367
248	11111000	F8	370
249	11111001	F9	371
250	11111010	FA	372
251	11111011	FB	373
252	11111100	FC	374
253	11111101	FD	375
254	11111110	FE	376
255	11111111	FF	377

Index for TRSDOS & DISK BASIC Reference Manual



INDEX

For: TRSDOS Version 2.1
DISK BASIC Version 1.1

Index

Subject	Page
&H, BASIC hex constant prefix	7-6
&O, BASIC octal constant prefix	7-6
<EN>, carriage return character	7-38
<LF>, line-feed character	7-38
access,	8-2
random	7-65, 8-11
sequential	7-60, 8-10
address	8-2
alphanumeric	8-2
argument	8-2
array	8-2
-notation	1-4
ASCII	8-2
format	7-32, 8-2
assembly-language	
I/O, TRSDOS	6-5
access from BASIC	7-14, 7-20
ATTRIB, TRSDOS command	4-12
AUTO, TRSDOS command	4-11
BACKUP, TRSDOS utility	
abbreviated instructions for	2-16
detailed description of	5-2
Important Notice	2-17
background task	8-2
backup disk	8-3
Base Conversions,	
decimal/binary/octal/hex	8-18
BASIC, TRSDOS	
command file	1-2, 3-4, 7-2
BASIC2, TRSDOS command	4-2
baud	8-3
binary	8-3
bit	8-3
blocking of logical records	
under TRSDOS	6-3, 6-7
bootstrap program	8-3
break	8-3
buffer	6-5, 6-6, 7-2, 7-33 ff, 8-3
byte	8-4
cassette I/O under	
DISK BASIC	7-5, 7-10
CLOCK, TRSDOS command	4-14
clock, real time, see real-time clock	
CLOSE, BASIC statement	7-36
close a file	6-11, 7-36, 8-4
CMD"D", BASIC statement	7-7
CMD"R", BASIC statement	7-10
CMD"S", BASIC statement	7-10
CMD"T", BASIC statement	7-10
command	
file	3-7, 8-4
format	3-5
command,	
TRSDOS library	4-2 ff, 8-8
TRSDOS system	4-11 ff
compressed format, BASIC	7-31, 8-4
COPY, TRSDOS command	4-15
CVD, BASIC function	7-54
CVI, BASIC function	7-54
CVS, BASIC function	7-54
data	8-4
data/device control block	
(DCB)	6-6, 8-4
data diskette	5-4
DATE, TRSDOS command	4-15
DEBUG, TRSDOS command	4-3
debug	8-4
decimal	8-5
default	8-5
delimiter,	6-6, 8-5
BASIC INPUT #	7-40
DEFFN, BASIC statement	7-11
DEFUSR, BASIC statement	7-14
destination	8-5
diskette	5-2
DEVICE, TRSDOS command	4-16
device	8-5
DIR, TRSDOS command,	4-16
directory	8-5

Subject	Page
DISK BASIC	
error messages	.7-77
ROM calls	.7-22
versions and releases	1-6
DISKDUMP/BAS, auxiliary utility program	.5-8 ff
diskette,	
data	.2-10
TRSDOS software	.2-10
diskette	.2-5 ff, 8-5
care	2-8
organization	2-6, 6-2
specifications	.2-10
drive number	8-6
drive numbering	2-5
drive specification	3-6, 3-8, 8-5
drive zero	2-5
dummy variable	8-6
DUMP, TRSDOS command	.4-18
end of file (EOF)	.4-16, 6-6, 6-9, 7-55, 8-6
entry point	.4-18, 5-6, 6-8, 7-14, 8-6
EOF, BASIC function	.7-55
ERROR, BASIC statement	7-6
error messages,	
BASIC	.7-6, 7-77
TRSDOS	.6-12
expression	8-6
FIELD, BASIC statement	.7-47
field,	8-6
overlapping	.7-76
field name	.7-48, 7-67, 8-6
file, TRSDOS	.3-3, 6-3 ff, 8-6
extension	.3-6 ff, 7-32, 8-6
name	3-6, 8-7
specification	.3-6 ff, 6-6, 8-7
foreground task	.3-2, 4-9, 4-10, 8-7
FORMAT, TRSDOS utility	.5-4 ff
format	.5-4 ff, 8-7
FREE, TRSDOS command	.4-19
GET, BASIC statement	.7-49
granule	.4-16, 4-19, 5-4, 6-3, 6-12, 8-7
hexadecimal	8-7
constants, BASIC	7-6
increment	8-7
input	8-7
INPUT #, BASIC statement	.7-37
INSTR, BASIC function	.7-15
interrupt	.7-5, 7-10, 8-8
KILL, BASIC command	.7-28
KILL, TRSDOS command	.4-19
kilobyte	8-8
LIB, TRSDOS command	.4-19
library command, TRSDOS	.4-11
LINE INPUT, BASIC statement	.7-16
LINE INPUT #, BASIC statement	.7-42
LIST, TRSDOS command	.4-20
LOAD, BASIC command	.7-28
LOAD, TRSDOS command	.4-20
LOF, BASIC function	.7-56
LSET, BASIC statement	.7-53
machine language,	8-8
dump to disk	.4-18
load from disk	.4-20
reserve RAM for	7-3
access routines from BASIC	.7-14, 7-20
Master Password	.4-21, 5-2
MERGE, BASIC command	.7-29
MID$=, BASIC statement	.7-17
Mini Disk	.2-1 ff
connection	2-3
operation	2-5
specifications	.2-10
MKD$, BASIC function	.7-57
MKI$, BASIC function	.7-57
MKS$, BASIC function	.7-57
Notation	1-3, 6-5
null string	8-8

Index

Subject	Page
object code	8-8 and see machine language
octal	8-9
constants, BASIC	7-6
OPEN, BASIC statement	.7-34
open a file	.6-6, 6-9, 7-33, 7-34, 8-9
parameter	8-9
password	.3-6, 3-8, 4-12, 4-13, 8-9
PRINT, TRSDOS command	.4-21
PRINT #, BASIC statement	.7-43
prompt	.8-10
PROT, TRSDOS command	.4-21
protected file	.6-12, 8-10
protection level	.4-12, 4-22, 8-10
PUT, BASIC statement	.7-50
random access	.7-65
random access memory (RAM)	.8-10
allocation	1-2, 3-4, 6-2
random access memory (RAM), BASIC program storage in	.7-32
reserving for machine language code under DISK BASIC	7-3
read only memory (ROM)	.8-10
calls from BASIC	.7-22
calls from TRSDOS	.6-8 ff
organization	6-2
real-time clock, in Expansion Interface	1-2
memory location of	4-9
to display	.4-14
to set	.4-23
to turn off	.7-10
to turn on	.7-10
record, TRSDOS physical	6-4, 6-7, 8-9
logical	6-7, 8-8
release	1-6
RENAME, TRSDOS command	.4-22
reset	.8-10
resident program	.3-4, 8-10
ROM (see read only memory)	
RSET, BASIC statement	.7-53
RUN"program", BASIC command	.7-31
SAVE, BASIC command	.7-31
sector, diskette	.2-6, 8-11
sequential access	.7-60, 8-11
statement, BASIC	.8-11
string	.8-11
subrecord	.7-71
syntax,	.8-11
TRSDOS command	3-5
system command	.4-22
file	.3-7, 8-11
routine	.6-5 ff
TAPEDISK, auxiliary utility	5-6
task, background	8-2
foreground	.3-2, 4-9, 4-10, 4-14, 8-7
TIME, TRSDOS command	.4-23
TIME$, BASIC function	.7-19
TRACE, TRSDOS command	.4-10
transfer address	.4-18, 8-11
track, diskette	.2-6, 8-11
TRSDOS assembly I/O	.6-5 ff
error messages	.6-12
file specification	3-6
library commands	.4-11
memory organization	6-2
RAM allocation	.3-4, 8-13
system commands	4-2
utilities	.5-1 ff
versions and releases	1-6
USING, BASIC PRINT format modifier	.7-46
USR, BASIC function	.7-20
utility TRSDOS	5-2
auxiliary	5-6
VERIFY, TRSDOS command	.4-24
versions and releases	1-6
write protect	2-6

IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

LIMITED WARRANTY

Radio Shack warrants for a period of 90 days from the date of delivery to customer that the computer hardware described herein shall be free from defects in material and workmanship under normal use and service. This warranty shall be void if this unit's case or cabinet is opened or if the unit is altered or modified. During this period, if a defect should occur, the product must be returned to a Radio Shack store or dealer for repair. Customer's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at Radio Shack's election and sole expense, except there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including but not limited to statements regarding capacity, suitability for use, or performance of the equipment, shall be or be deemed to be a warranty or representation by Radio Shack, for any purpose, nor give rise to any liability or obligation of Radio Shack whatsoever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

RADIO SHACK TC **A DIVISION OF TANDY CORPORATION**

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.

Name: _____

Company: _____

Mailing Address: _____

_____

Phone: _____

Brief list of hardware in your system:

_____

_____

_____

_____

Please help us better serve your needs – answer these survey questions and return the survey to us as as soon as possible.

1. What do/will you use your TRS-80 TRSDOS System for? _____

_____

2. What is your approximate investment in hardware $_____ and software $ _____ ?

3. What area(s) of your TRS-80 system needs the most improvement? _____

_____

_____

_____

_____

_____

4. What approach would be the most effective for accomplishing those improvements? _____

_____

_____

_____

_____

_____

5. What would be the result if these improvements were made? _____

_____

_____

_____

Thank you.

(Fold Along This Line)

Place
Stamp
Here

TRS-80 Computer Survey
1100 One Tandy Center
Fort Worth, TX 76102

(Fold Along This Line)